AFRL-RI-RS-TR-2008-273
Final Technical Report
October 2008

EVALUATING SPARSE LINEAR SYSTEM SOLVERS ON SCALABLE PARALLEL ARCHITECTURES

Purdue University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AD53

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-273 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
CHRISTOPHER FLYNN Work Unit Manager	EDWARD J. JONES Deputy Chief, Advanced Computing Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
OCT 08	Final	Aug 06 – May 08

4. TITLE AND SUBTITLE

EVALUATING SPARSE LINEAR SYSTEM SOLVERS ON SCALABLE PARALLEL ARCHITECTURES

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-06-1-0233

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Ananth Grama, Murat Manguoglu, Mehmet Koyuturk, Maxim Naumov and Ahmed Sameh

5d. PROJECT NUMBER
AD53

5e. TASK NUMBER
PR

5f. WORK UNIT NUMBER
DU

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

PRIME
Purdue University
Dept of Computer Science
West Lafayette IN 47907-2024

SUB
Case Western Reserve University
Dept of EE and Computer Science
Cleveland OH 44106

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/RITB
525 Brooks Rd
Rome, NY 13441-4515

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-273

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 08-0519

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes in detail studies developing and evaluating sparse linear systems on scalable architectures, with emphasis on preconditioned iterative solvers. The study was motivated primarily by the lack of robustness of Krylov subspace iterative schemes with generic, "black-box, pre-conditioners such as approximate (or incomplete) LU-factorizations. In this report the authors advocate the use of banded pre-conditioners after suitable reordering of the sparse linear systems. The choice of the reordering scheme is based on: 1) minimizing the bandwidth, and 2) bringing as many of the largest elements of the coefficient matrix as possible to a "narrow" central band.

15. SUBJECT TERMS
Sparse linear systems solvers, scalability, parallel architectures

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Christopher Flynn
U	U	U	UU	104	19b. TELEPHONE NUMBER (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

ABSTRACT

This report describes in detail our studies in developing and evaluating sparse linear systems on scalable architectures, with emphasis on preconditioned iterative solvers. The study was motivated primarily by the lack of robustness of Krylov subspace iterative schemes with generic, "black-box", preconditioners such as approximate (or incomplete) LU-factorizations as well as their limited scalability to large-scale parallel platforms. In this report we advocate the use of banded preconditioners after suitable reordering of the sparse linear systems. The choice of the reordering scheme is based on: (i) minimizing the bandwidth, and (ii) bringing as many of the largest elements of the coefficient matrix as possible to a "narrow" central band. Next, we extract a prominent central band and use it as a preconditioner. In Chapters 1 and 2 we develop a parallel algorithm, "SPIKE", for solving banded systems that are considered dense within the band. We also show that our solver is highly scalable and superior in performance to the banded system solver in the widely available parallel library ScaLapack. Also, we show how "SPIKE" can be used in conjunction with a direct sparse system solver such as "Pardiso", which is a subroutine in Intel's Math Kernel Library (MKL) for handling banded systems that are sparse within the band. In Chapter 3, we present a weighted reordering scheme that enables the extraction of effective banded preconditioners. We also show that the time consumed by such a reordering scheme is a small fraction of the total time needed to solve very large sparse systems using preconditioned BiCGstab or GMRES (two prominent members of the Krylov subspace methods). The effectiveness of our strategy is demonstrated on a wide collection of publicly available sparse linear systems at the University of Florida (see Tim Davis' web-site). In Chapter 4, we describe an alternate scalable parallel algorithm for handling banded preconditioners that arise from domain decomposition in which the domains overlap. We demonstrate the effectiveness of this solver compared to direct solvers such as ScaLapack or preconditioned iterative solvers for both symmetric positive definite systems as well as non-symmetric systems. Finally, in Chapter 5, we present our study concerning prediction of parallel scalability of our schemes on architectures with more nodes/cores than the platforms on which our experiments have been performed.

Chapter 1

Highly Scalable Linear Solvers: The SPIKE Algorithm

1.1 Introduction

Banded linear systems arise in many areas of computational science and engineering such as computational mechanics (fluids, structures, and fluid-structure interaction) [69, 71] and computational nanoelectronics [59]. These applications often give rise to very large narrow banded linear systems, which can be dense or sparse within the band. For example, in finite-element analysis, the underlying sparse linear systems can be reordered to result in a banded system in which the width of the band is a small fraction of the number of unknowns.

In this report, we describe important algorithmic and performance-related aspects of Spike, [9, 10, 33, 51, 65, 66, 67, 68, 70]. The underlying basis for Spike is a divide and conquer technique, which involves the following stages: (a) pre-processing: (i) partitioning of the original system on different processors, or different Symmetric Multiprocessors (SMP's), (ii) factorization of each diagonal block and extraction of a reduced system of much smaller size; (b) post-processing: (iii) solving the reduced system, and (iv) retrieving the overall solution. Not only does SPIKE enable multilevel parallelism, but it also allows multiple choices for the pre- and post-processing stages, resulting in a poly-algorithm. We will show that this algorithm has several built-in options that range from using it as a pure direct solver to using it as a preconditioner for any iterative scheme.

We present several numerical experiments that demonstrate significant speed and scalability improvements over corresponding routines in ScaLapack [12] for handling large banded systems on a high-end computing platform.

A general description of the SPIKE algorithm is presented in Section 1.2.

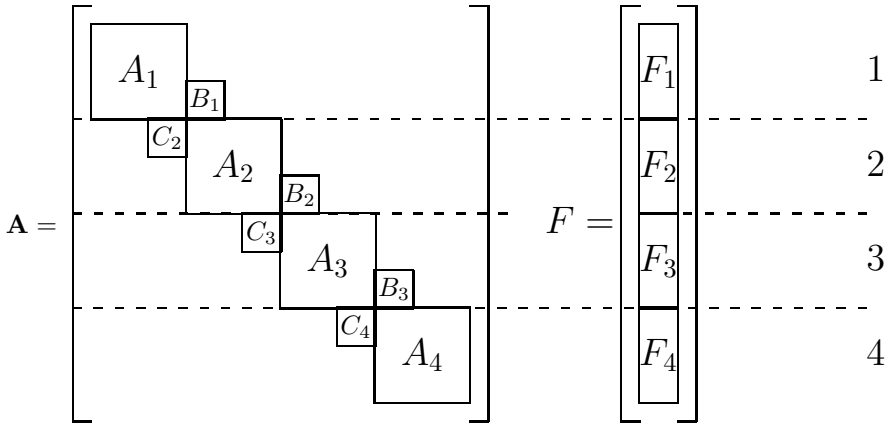


Figure 1.1: *Partitioning of the matrix A and the RHS F, with $p = 4$. The size of each partition j is n_j, and the size of the coupling off-diagonal blocks B_j and C_j of the original matrix, is $m \times m$.*

The hybrid and poly-algorithm nature of SPIKE is discussed in Section 1.3 along with different options for the treatment of the pre- and post-processing stages. In particular, Section 1.4 and 1.5 focus on the detailed description of two versions of the algorithm: the "Recursive" and the "Truncated" schemes, for handling general and diagonally dominant systems, respectively. In Section 1.6, we present several numerical experiments with performance comparisons versus ScaLapack on the computing platform IBM-SP.

1.2 The Spike algorithm

Consider solving the linear systems $\mathbf{AX} = \mathbf{F}$, where $\mathbf{A}$ is a narrow banded $n \times n$ matrix, and $\mathbf{F}$ is the $n \times s$ multiple right hand side (RHS). We assume that the bandwidth b is much less than the system order n.

1.2.1 Preprocessing stage

Partitioning

Any banded linear system can be partitioned into a block tridiagonal form. If p is the number of diagonal blocks, then each banded diagonal block A_j ($j = 1, \ldots, p$), is of order n_j (or roughly of order n/p). For a given partition j, B_j ($j = 1, \ldots, p-1$), and C_j ($j = 2, \ldots, p$), are the coupling matrices associated with the first super- and sub-diagonal blocks, respectively. Each is of order $m << n_j$. Figure 1.1 illustrates the partitioning of the matrix $\mathbf{A}$ and the RHS for $p = 4$.

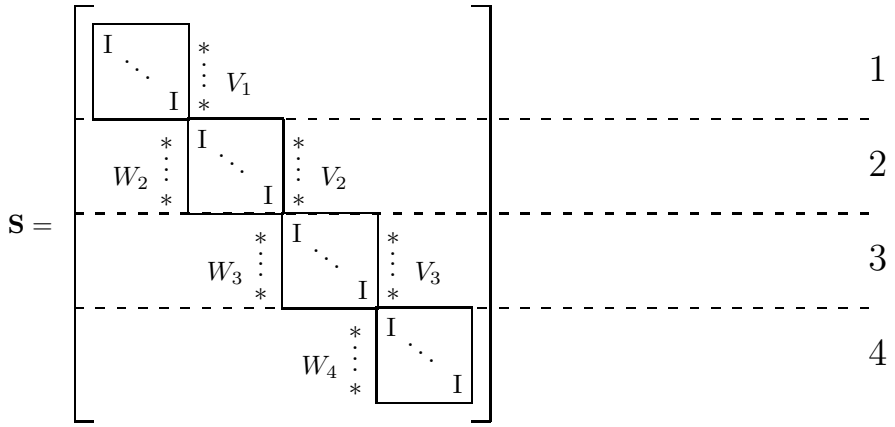


Figure 1.2: *The spike factorization defined by* $\mathbf{A} = \mathbf{DS}$*, where* $\mathbf{D}$ *is a block-diagonal matrix with* 4 *partitions. A new linear system* $\mathbf{SX} = \mathbf{G}$ *needs, then, to be solved, where* $\mathbf{G}$ *is the modified right hand side (*$\mathbf{DG}) = \mathbf{F}$*). An independent reduced system, of much smaller size, can be extracted from those few rows of* $\mathbf{S}$ *immediately above and below each partitioning line.*

Each partition can be associated with one or several processors (one node), enabling multilevel parallelism. For structurally symmetric problems, m is equal to $(b-1)/2$, while for non-symmetric structure, for coding convenience, we include some zero elements either in B_j or C_j to maintain structural symmetry, resulting in $m > (b-1)/2$.

1.2.2 Factorization

The preprocessing stage includes the factorization of each diagonal block, and obtaining a new system that results from pre-multiplying each partition by the inverse of its diagonal block. The resulting new system has identity matrices as diagonal blocks, and spikes of m columns in the immediate off-diagonal blocks.

Assuming, for the time being, that each A_j is nonsingular, the matrix can be factored as $\mathbf{A} = \mathbf{D}\ \mathbf{S}$, where $\mathbf{D}$ is a block-diagonal matrix consisting only of the diagonal blocks A_j,

$$\mathbf{D} = \mathrm{diag}(A_1, ..., A_p),$$

with $\mathbf{S}$ being the spike matrix shown in Fig. 1.2. For a given partition j, we call V_j $(j = 1, \ldots, p-1)$ and W_j $(j = 2, \ldots, p)$, respectively, the right and the left spikes each of order $n_j \times m$.

The spikes V_j and W_j are given by

$$V_j = (A_j)^{-1} \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j, \quad \text{and} \quad W_j = (A_j)^{-1} \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j. \tag{1.1}$$

The spikes V_j and W_j, $j = 2, ..., p-1$, may be generated by solving,

$$A_j \left[V_j, W_j \right] = \begin{bmatrix} 0 & C_j \\ \vdots & 0 \\ 0 & \vdots \\ B_j & 0 \end{bmatrix}. \tag{1.2}$$

1.2.3 The Post-processing stage

Solving the system $\mathbf{AX} = \mathbf{F}$ now reduces to two steps:

$$\text{(a) solve} \quad \mathbf{DG} = \mathbf{F} \tag{1.3}$$

$$\text{(b) solve} \quad \mathbf{SX} = \mathbf{G}. \tag{1.4}$$

The solution of the linear system $\mathbf{DG} = \mathbf{F}$ in Step (a), yields the modified RHS $\mathbf{G}$ needed for Step b, and in case of one partition per processor, is performed with perfect parallelism. In case we decouple the pre-and post-processing stages, this step may be combined with the generation of the spikes in equations (1.2).

To solve $SX = G$ in Step b, one should observe that the problem can be reduced further by solving an independent system of much smaller size,

$$\hat{\mathbf{S}}\hat{\mathbf{X}} = \hat{\mathbf{G}}, \tag{1.5}$$

which consists of the m rows of $\mathbf{S}$ immediately above and below each partitioning line. Indeed, the spikes V_j and W_j can also be partitioned as follows

$$V_j = [V_j^{(t)} \quad V_j' \quad V_j^{(b)}]^T, \quad \text{and} \quad W_j = [W_j^{(t)} \quad W_j' \quad W_j^{(b)}]^T, \tag{1.6}$$

where $V_j^{(t)}$, V_j', $V_j^{(b)}$, and $W_j^{(t)}$, W_j', $W_j^{(b)}$, are the top m, the middle $n_j - 2m$ and the bottom m rows of V_j and W_j, respectively. We note that

$$V_j^{(b)} = [0 \quad I_m] V_j; \quad W_j^{(t)} = [I_m \quad 0] W_j, \tag{1.7}$$

and

$$V_j^{(t)} = [I_m \quad 0] V_j; \quad W_j^{(b)} = [0 \quad I_m] W_j. \tag{1.8}$$

Similarly, if X_j and G_j are the jth partitions of $\mathbf{X}$ and $\mathbf{G}$, we have

$$X_j = [X_j^{(t)} \quad X_j' \quad X_j^{(b)}]^T, \quad \text{and} \quad G_j = [G_j^{(t)} \quad G_j' \quad G_j^{(b)}]^T. \tag{1.9}$$

It is then possible to extract, from (1.4), the independent reduced linear system (1.5), which involves only the top and bottom elements of V_j, W_j, X_j and G_j.

This reduced system is block tridiagonal, with $(p-1)$ diagonal blocks, the kth of which is given by

$$\begin{bmatrix} I_m & V_k^{(b)} \\ W_{k+1}^{(t)} & I_m \end{bmatrix}. \tag{1.10}$$

The corresponding off-diagonal blocks are

$$\begin{bmatrix} W_k^{(b)} & 0 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 0 \\ 0 & V_{k+1}^{(t)} \end{bmatrix}, \tag{1.11}$$

and the associated solution and RHS are

$$[X_k^{(b)} \quad X_{k+1}^{(t)}]^T, \quad \text{and} \quad [G_k^{(b)} \quad G_{k+1}^{(t)}]^T.$$

Once the solution $\hat{\mathbf{X}}$ of the reduced system (1.5) is obtained, the global solution $\mathbf{X}$ is reconstructed with perfect parallelism from X_k^b $(k = 1, \ldots, p-1)$ and X_k^t $(k = 2, \ldots, p)$ as follows:

$$\begin{cases} X_1' = G_1' - V_1' X_2^{(t)}, \\ X_j' = G_j' - V_j' X_{j+1}^{(t)} - W_j' X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1 \\ X_p' = G_p' - W_j' X_{p-1}^{(b)}. \end{cases} \tag{1.12}$$

1.3 SPIKE: a poly-algorithm

Several options are available for efficient implementation of the SPIKE algorithm on parallel architectures, for reducing the complexity and required storage. These choices depend on the properties of the linear system, as well as the architecture at hand. More specifically, SPIKE allows three major options:

1. factorization of the diagonal blocks A_j,
2. computation of the spikes, and
3. solution scheme for handling the reduced system.

In the first option, each linear system associated with A_j, can be solved either (i) directly, making use of Gaussian elimination (LU-factorization) with partial pivoting or Cholesky factorization if (A_j) is symmetric positive definite, (ii) using LU factorization without pivoting but with a diagonal boosting strategy, (iii) iteratively with a preconditioning strategy, (iv) or via appropriate approximation of the inverse of (A_j). If more than one processor is associated with one partition, another level of the SPIKE algorithm may also be used. In

the following, we consider only the case where one partition is associated with only one processor.

In the second option, the spikes can be computed either explicitly (fully or partially) using equation (1.2), or implicitly – "on-the-fly". We will restrict the treatment here to explicit partial or complete determination of the spikes.

In the third option, the reduced system (1.5) can be solved either (i) directly using a "recursive" form of the SPIKE algorithm, (ii) iteratively with a preconditioning scheme, or (iii) approximately within a "truncated" SPIKE scheme for diagonally dominant systems. Both the recursive and truncated algorithms will be presented in detail in the following sections.

In addition, outer iterations will be necessary to assure sufficient accuracy whenever we do not use a direct method to solve (1.3) or (1.5). The overall SPIKE algorithm is then used as a preconditioner for the outer iterative scheme (Krylov subspace scheme, or iterative refinement) in which (1.3) is solved once and for all.

A multitude of options for SPIKE have been implemented. However, a complete description of implementation details is beyond the scope of this report. In the following we describe two optimal algorithms for handling general and diagonally dominant systems.

1.4 The non-diagonally dominant case

In this section, we present a version of the SPIKE algorithm that is general enough for handling non-diagonally dominant systems. In particular, we consider LU factorization of the diagonal blocks without pivoting, but with a "diagonal boosting" strategy to obtain an LU factorization of slightly perturbed counterparts of the original block. This is followed by an efficient recursive SPIKE scheme for solving the resulting reduced system.

1.4.1 Factorization step

For numerical stability, one obtains the LU factorization of each block A_j with partial pivoting, often by making use of the LAPACK routines [1] XGBTRF. Here X corresponds to S, D, or Z, for single real, double real, and double complex arithmetic. The LAPACK routines XGBTRS, based on Basic Linear Algebra Subroutines BLAS level-2 [1], may be used to solve for the spikes (1.2) and the RHS (1.3). We have modified this LAPACK procedure in order to take advantage of BLAS level-3, and used it to solve triangular systems that arise in SPIKE.

We have also developed a version of SPIKE in which the LU-Factorization is performed without pivoting but with a "diagonal boosting" strategy to overcome problems associated with very small pivots. The magnitude of the pivot must then satisfy the following condition:

$$|\text{pivot}| > 0_\epsilon \; ||A_j||,$$

where 0_ϵ is a newly defined "machine zero", and $||.||$ is the 1-norm, for example. If the pivot does not satisfy this condition, then its value is boosted using ϵ, which depends on the machine's unit roundoff:

$$\begin{aligned} \text{pivot} &= \text{pivot} + \epsilon||A_j|| \quad \text{if pivot} > 0, \\ \text{pivot} &= \text{pivot} - \epsilon||A_j|| \quad \text{if pivot} < 0. \end{aligned}$$

We have implemented a modified version of the XDBTRF routine (originally developed for the ScaLapack package to handle diagonally dominant matrices) augmented by a diagonal boosting strategy. If diagonal boosting is activated, SPIKE is not used as a direct solver but rather as a preconditioner. In this case outer iterations via a Krylov subspace method, for example, are activated. In most cases, few outer iterations are needed to achieve adequate reduction of the relative residuals. Finally, since the LU factorization is performed without pivoting, our BLAS3-based block triangular solver can be used for both the forward and backward sweeps for solving (1.2) and (1.3).

1.4.2 The Recursive SPIKE algorithm

One natural way to solve the reduced system in parallel is to make use of a Krylov subspace iterative method with a block Jacobi preconditioner defined by the diagonal blocks (1.10) of the reduced system. However, for non-diagonally dominant systems, this preconditioner may not be effective for large number of partitions (producing large reduced systems). This often results in a large number of iterations if one is to realize reasonable residuals, and in high interprocessor communication cost. If the unit cost of interprocessor communication is high, the reduced system may be solved directly on a single processor. This alternative, however, may have memory limitations if the size of the reduced system is large.

We propose a new direct scheme for solving the reduced system in parallel. This "Recursive" scheme, involves successive iterations of the SPIKE algorithm resulting in better balance between the computational and communication costs.

Preliminary: The two-partitions case

Using only two partitions, $p = 2$, one can observe that the reduced system, which consists now of only one diagonal block (1.10), needs only to be solved directly. This reduced system can be extracted from the central part of the system (1.4)

$$\begin{bmatrix} I_m & V_1^{(b)} \\ W_2^{(t)} & I_m \end{bmatrix} \begin{bmatrix} X_1^{(b)} \\ X_2^{(t)} \end{bmatrix} = \begin{bmatrix} G_1^{(b)} \\ G_2^{(t)} \end{bmatrix}. \tag{1.13}$$

The solution steps consist of the following:

- Form $E = I_m - W_2^{(t)} V_1^{(b)}$

- Solve $EX_2^{(t)} = G_2^{(t)} - W_2^{(t)} G_1^{(b)}$ to obtain $X_2^{(t)}$
- Compute $X_1^{(b)} = G_1^{(b)} - V_1^{(b)} X_2^{(t)}$.

The rest of the solution of X_1, and X_2 can be retrieved using (1.12).

Recursive SPIKE: the multiple partitions case

Here, we assume that the number of partitions is given by $p = 2^d$ ($\forall d > 1$). After forming the spike matrix $\mathbf{S}$ (see Fig. 1.2), the number of partitions of the new linear system (1.4) can be divided by two, and another level of the SPIKE algorithm may be applied. This process is repeated recursively until we obtain only two partitions for the newest matrix $\mathbf{S}$. The resulting reduced system has the form (1.13).

In our implementation, the Recursive scheme is not concerned with the overall matrix $\mathbf{S}$ but rather with the matrix $\hat{\mathbf{S}}$ of the reduced system (1.5) itself. This allows us to simplify the implementation, and reduce the memory requirements while saving all the different levels of the new spikes. Note that in the reduced system (1.5), the matrix $\hat{\mathbf{S}}$ is block tridiagonal, where the diagonal blocks are defined in (1.10) and the off diagonal blocks are as given in (1.11).

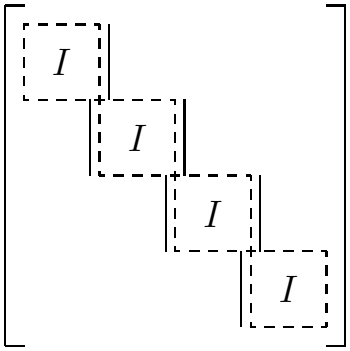
Observing that we can also extract an independent reduced system of order $2mp$ rather than order $2m(p-1)$, if we include, in addition, the top m rows and the bottom m rows of the first and last partitions, respectively. In this case, the structure of the reduced system remains block tridiagonal in which each diagonal block is an identity matrix of order $2m$, and the off-diagonal blocks associated with the k-th diagonal block are given by,

$$\begin{bmatrix} 0 & W_k^{(t)} \\ 0 & W_k^{(b)} \end{bmatrix} \text{ for } k = 2, \ldots, p \ , \quad \text{and} \quad \begin{bmatrix} V_k^{(t)} & 0 \\ V_k^{(b)} & 0 \end{bmatrix} \text{ for } k = 1, \ldots, p-1. \tag{1.14}$$

Denoting the spikes of the new reduced system at level 1 of the recursion by $v_k^{[1]}$ and $w_k^{[1]}$, where

$$v_k^{[1]} = [V_k^{(t)} \quad V_k^{(b)}]^T, \quad \text{and} \quad w_k^{[1]} = [W_k^{(t)} \quad W_k^{(b)}]^T, \tag{1.15}$$

the matrix $\tilde{\mathbf{S}}_{\mathbf{1}}$ of the new reduced system, for $p = 4$ takes the form:

$\tilde{\mathbf{S}}_{\mathbf{1}} =$ 

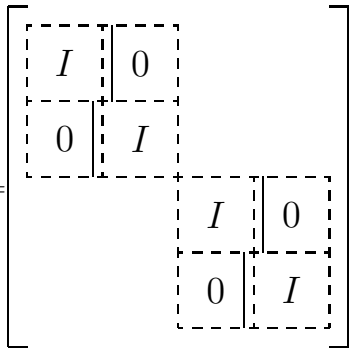
right spikes: $v_k^{[1]}$; $k = 1, 2, 3$

left spikes: $w_k^{[1]}$; $k = 2, 3, 4$.

In preparing for level 2 of the recursion of the Spike algorithm, we partition the matrix $\tilde{\mathbf{S}}_{\mathbf{1}}$ using $p/2$ partitions each of size $4m$. The matrix $\tilde{\mathbf{S}}_{\mathbf{1}}$ can then be factored as

$$\tilde{\mathbf{S}}_{\mathbf{1}} = \mathbf{D}_{\mathbf{1}}\ \tilde{\mathbf{S}}_{\mathbf{2}}$$

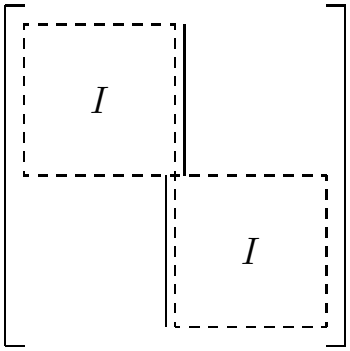
where D_1 is formed by the $p/2$ diagonal block of $\tilde{\mathbf{S}}_{\mathbf{1}}$ each of size $4m$, thus $\tilde{\mathbf{S}}_{\mathbf{2}}$ represents the new Spike matrix at the level 2 composed of the spikes $v_k^{[2]}$ and $w_k^{[2]}$. For $p = 4$, these matrices are of the form,

$\mathbf{D}_{\mathbf{1}} =$ 

right spikes: $v_k^{[2]}$; $k = 1, 3$

left spikes: $w_k^{[2]}$; $k = 2, 4$

and

$\tilde{\mathbf{S}}_{\mathbf{2}} =$ 

right spikes: $v_k^{[3]}$; $k = 1$

left spikes: $w_k^{[3]}$; $k = 2$.

In general, at level i of the recursion, the spikes $v_k^{[i]}$, and $w_k^{[i]}$, with k ranging from 1 to $p/(2^i)$, are of order $2^i m \times m$. Thus, if the number of the original partitions p is equal to 2^d, the total number of recursion levels is $d-1$ and the matrix $\tilde{\mathbf{S}}_{\mathbf{1}}$ can be expressed in the factored form,

$$\tilde{\mathbf{S}}_{\mathbf{1}} = \mathbf{D_1 D_2} \dots \mathbf{D_{d-1}}\ \tilde{\mathbf{S}}_{\mathbf{d}}$$

where the matrix $\tilde{\mathbf{S}}_{\mathbf{d}}$ has only two spikes $v_1^{[d]}$ and $w_2^{[d]}$. The reduced system can then be written as

$$\tilde{\mathbf{S}}_{\mathbf{d}} \tilde{\mathbf{X}} = \mathbf{B}, \tag{1.16}$$

where $\mathbf{B}$ is the modified right hand side:

$$\mathbf{B} = \mathbf{D_{d-1}^{-1}} \dots \mathbf{D_2^{-1} D_1^{-1}}\ \tilde{\mathbf{G}}. \tag{1.17}$$

If we assume that the spikes $v_k^{[i]}$, and $w_k^{[i]}$ $(\forall k)$ of the matrix $\tilde{\mathbf{S}}_{\mathbf{i}}$ are known at a given level i, then we can compute the spikes $v_k^{[i+1]}$, and $w_k^{[i+1]}$ at level $i+1$ as follows:

Step 1. Denoting the bottom and the top blocks of the spikes at the level i by

$$v_{k'}^{[i](b)} = [0 \quad I_m] v_{k'}^{[i]}; \quad w_{k'}^{[i](t)} = [I_m \quad 0] w_{k'}^{[i]},$$

and the middle block of $2m$ rows of the spike at level $i+1$ by

$$\begin{bmatrix} \dot{v}_k^{[i+1]} \\ \ddot{v}_k^{[i+1]} \end{bmatrix} = [0 \quad I_{2m} \quad 0] v_k^{[i+1]}; \quad \begin{bmatrix} \dot{w}_k^{[i+1]} \\ \ddot{w}_k^{[i+1]} \end{bmatrix} = [0 \quad I_{2m} \quad 0] w_k^{[i+1]},$$

one can form the following reduced systems

$$\begin{bmatrix} I_m & v_{2k-1}^{[i](b)} \\ w_{2k}^{[i](t)} & I_m \end{bmatrix} \begin{bmatrix} \dot{v}_k^{[i+1]} \\ \ddot{v}_k^{[i+1]} \end{bmatrix} = \begin{bmatrix} 0 \\ v_{2k}^{[i](t)} \end{bmatrix}, \quad \forall k = 1, 2, \dots, \frac{p}{2^{i-1}} - 1, \tag{1.18}$$

and

$$\begin{bmatrix} I_m & v_{2k-1}^{[i](b)} \\ w_{2k}^{[i](t)} & I_m \end{bmatrix} \begin{bmatrix} \dot{w}_k^{[i+1]} \\ \ddot{w}_k^{[i+1]} \end{bmatrix} = \begin{bmatrix} w_{2k-1}^{[i](b)} \\ 0 \end{bmatrix} \quad \forall k = 2, 3, \ldots, \frac{p}{2^{i-1}}. \tag{1.19}$$

These reduced systems are solved in a manner similar to (1.13) to obtain the solutions of the center parts of the spikes at the level $i+1$.

Step 2 The entire solution of the spikes at the level $i+1$ is retrieved as follows

$$[I_{2^i m} \quad 0] v_k^{[i+1]} = -v_{2k-1}^{[i]} \ddot{v}_k^{[i+1]}, \qquad [0 \quad I_{2^i m}] v_k^{[i+1]} = v_{2k}^{[i]} - w_{2k}^{[i]} \dot{v}_k^{[i+1]}, \tag{1.20}$$

and

$$[I_{2^i m} \quad 0] w_k^{[i+1]} = w_{2k-1}^{[i]} - v_{2k-1}^{[i]} \ddot{w}_k^{[i+1]} \qquad [0 \quad I_{2^i m}] w_k^{[i+1]} = -w_{2k}^{[i]} \dot{w}_k^{[i+1]}. \tag{1.21}$$

In order to compute one step of the modified RHS (1.17) as $\tilde{\mathbf{G}}_\mathbf{i} = \mathbf{D}_\mathbf{i}^{-\mathbf{1}} \tilde{\mathbf{G}}_{\mathbf{i-1}}$ (with the first being $\tilde{\mathbf{G}}_\mathbf{1} = \mathbf{D}_\mathbf{1}^{-\mathbf{1}} \tilde{\mathbf{G}}$), one has to solve the linear system $\mathbf{D_i}\ \tilde{\mathbf{G}}_\mathbf{i} = \tilde{\mathbf{G}}_{\mathbf{i-1}}$, which is block diagonal. For each diagonal block k, the reduced systems are similar to those in (1.13) or in (1.18) and (1.19) but the RHS is now defined as function of $\tilde{\mathbf{G}}_{\mathbf{i-1}}$. Once we obtain the partial solution at the center part of each $\tilde{\mathbf{G}}_\mathbf{i}$, associated with each block k in $\mathbf{D_i}$, the entire solution is retrieved as in (1.20) and (1.21). In the same way, the linear system (1.16) involves only one reduced system to solve and only one retrieval stage to get the solution $\tilde{\mathbf{X}}$. Finally, the overall solution $\mathbf{X}$ is obtained using the procedure (1.12).

1.4.3 Additional remarks

The generation of the spikes at the various levels is included in the factorization step. In this way, the solver makes use of the spikes stored in the memory, thus allowing solution of the reduced system quickly and efficiently. Since in many applications one has to solve many linear systems with the same coefficient matrix $\mathbf{A}$ but with different RHSs, optimization of the solver step is thus crucial for the success of SPIKE.

1.5 The diagonally dominant case

In this section, we describe a version of the SPIKE algorithm that is optimized for handling diagonally dominant systems. A truncated scheme is proposed that yields a modified reduced system that is block diagonal rather than block tridiagonal. This allows solving the reduced system more efficiently on parallel architectures. Furthermore, it facilitates different new options for the factorization steps that make it possible to avoid the computation of the entire spikes.

1.5.1 The truncated SPIKE algorithm

If the matrix A_j is diagonally dominant, one can show that the magnitude of the elements of the right spikes V_j decay in magnitude from bottom to top, while the elements of the left spikes W_j decay from top to bottom. Since the size n of A_j is much larger than the size m of the blocks B_j and C_j, the bottom blocks of the left spikes $W_j^{(b)}$ and the top blocks of the right spikes $V_j^{(t)}$ can be approximately set equal to zero. Thus, it follows that the off-diagonal blocks of reduced system (1.11) are equal to zero, and the reduced matrix $\hat{\mathbf{S}}$ is then approximated by its block diagonal (1.10). Each diagonal block of this truncated reduced system can be solved directly using the same procedure described for solving the system (1.13), leading to enhanced use of parallelism.

1.5.2 The truncated factorization stage

Since the matrix is diagonally dominant, the LU factorization of each block A_j can be performed without pivoting, using for example the modified LAPACK routine XDBTRF in the ScaLapack package (no boosting strategy is necessary). Using an LU-factorization without pivoting, one can get the bottom block of V_j from (1.1) involving only the bottom $m \times m$ blocks of L and U. Obtaining the top block of W_j still requires computing the entire left j-th spike with complete forward and backward sweeps. Another approach consists of performing a UL-factorization (Gaussian elimination that produces the factorization in the form of the product of an upper triangular matrix and a lower triangular matrix) without pivoting. Similar to the LU-factorization, this allows obtaining the top block of W_j involving only the top $m \times m$ blocks of the new U, L. Our numerical experiments indicate that the time consumed by this LU/UL strategy, is much less than that taken by performing only one LU factorization per diagonal block and generating the entire left spikes. Using a permutation of the rows and columns of the original matrix, we can use the same LU factorization routine to perform the UL-factorization.

Approximation of the factorization stage

An alternate to performing a UL-factorization of the block A_j is to approximate the top of the left spike, $W_j^{(t)}$, of order m, by inverting only an $l \times l$ ($l > m$) top diagonal block (left top corner) of the banded block A_j. Typically, we choose $l = 2m$ to get a suitable approximation of the $m - by - m$ left top corner of the inverse of A_j. However, the quality of this approximation depends on the degree of diagonal dominance. With such a strategy, SPIKE thus needs to be used as a preconditioner. In general, convergence of an outer Krylov subspace iterative scheme is often realized after only few iterations.

Using the above Truncated scheme, the entire spikes are not computed explicitly. Therefore, once the reduced system is solved, retrieving of the entire solution cannot be done using equation (1.12). Rather, we extract the RHS from the

contributions of the tying blocks B_j and C_j, and then solve the resulting block diagonal system using either the previously computed LU or UL factorizations, as shown below:

$$\begin{cases} A_1\ X_1 = F_1 - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_2^{(t)}, \\ A_j\ X_j = F_j - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_{j+1}^{(t)} - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1 \\ A_p\ X_p = F_p - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{p-1}^{(b)}. \end{cases} \tag{1.22}$$

1.6 Performance results and comparisons with ScaLapack

The performance of different versions of the SPIKE algorithm are compared with ScaLapack's banded solvers. We consider two sets of experiments:

1. speed improvement over ScaLapack for systems with varying bandwidths on 32 processors.
2. speed improvement over ScaLapack on a given banded system as the number of processors is varied from 32 to 512.

All the tests are performed on the DataStar IBM-SP power4 platform. This platform has 176 (8-way) P655+ nodes, with 16 GB of memory and 1.5GHz CPU speed. Also, the use of each 8-way node is exclusive allowing only one user at the node at any time.

1.6.1 SPIKE: performance on 32 processors

We consider a system of order $n = 480,000$ that is dense within the band, with one RHS ($s = 1$). Three main tests with different SPIKE algorithms, have been conducted and the results are listed in Table (1.1). Tests 1, 2a and 2b, are concerned with non-diagonally dominant systems, while Tests 3a and 3b, deal with diagonally dominant ones as the bandwidth (maximum number of nonzero elements per row) varies from 81 to 641. These systems are solved using the SPIKE schemes presented, respectively, in sections 1.4 and 1.5. For each test, we show the speed improvement of the SPIKE algorithm over ScaLapack, $\lambda = \mathrm{T_{sca.}}/\mathrm{T_{Spike}}$, for the factorization, the solver, as well as the total time. The ScaLapack routines used for the factorization and solve stages, are PDGBTRF and PDGBTRS for non-diagonally dominant systems, and PDDBTRF and PDDBTRS for the diagonally dominant one.

Table 1.1: *List of various tests associated with various options for the SPIKE algorithm.*

	Diagonally Dominant	Factorization step	Solve step	Outer-iterations
Test 1	No	LU with pivoting	Recursive	No
Test 2a	No	LU without pivoting + boost.	Recursive	No
Test 2b	No	LU without pivoting + boost.	Recursive	Yes
Test 3a	Yes	LU-UL without pivoting	Truncated	No
Test 3b	Yes	LU without pivoting + approx.	Truncated	Yes

Test 1

In this experiment for non-diagonally dominant systems, SPIKE is performed using the Recursive algorithm to solve the reduced system. The LU-factorization of the diagonal blocks of the original matrix, is implemented with partial pivoting using the LAPACK routine: DGBTRF. SPIKE is thus used as a direct solver. Here, we have assumed that each diagonal block is nonsingular. The speed improvement factor, λ, presented in Fig.1.3 shows that SPIKE is, on the average twice as fast as ScaLapack for the factorization and solve steps.

Test 2a and 2b

In this set of numerical experiments, we assume that we have no knowledge regarding the non-singularity of the diagonal blocks. Thus, avoiding pivoting in the LU-factorization of the diagonal blocks, together with adopting a diagonal boosting strategy, we obtain LU factorizations of slightly perturbed block diagonal matrices. The Recursive SPIKE scheme is thus used as a preconditioner and outer-iterations are needed to assure convergence to the true solution, i.e., assuring that the relative residual:

$$r = \frac{||\mathbf{AX} - \mathbf{F}||_\infty}{||\mathbf{F}||_\infty},$$

reaches a given tolerance. If diagonal boosting is performed during the factorization stage (i.e., a "zero-pivot" is detected), the solve part of the SPIKE algorithm is automatically used inside an iterative scheme. In our current implementation, we can use either the BiCGstab iterative scheme, or regular iterative refinement. When an outer iteration is activated, we use the stopping criterion, $r < 10^{-8}$. If no diagonal boosting is performed during the factorization stage, no outer iteration is needed and SPIKE is used as a direct solver. Throughout, our experiments yielded relative residuals much smaller than 10^{-8}.

Figure 1.4 shows performance results for the two following cases: (i) Test 2a - no diagonal boosting, no outer-iteration needed, (ii) Test 2b - diagonal boosting, outer-iterations needed.

For the factorization stage of both Test 2a and Test 2b, the speed improvement λ over ScaLapack, starts at 2.1 for a bandwidth of $b = 81$, increasing

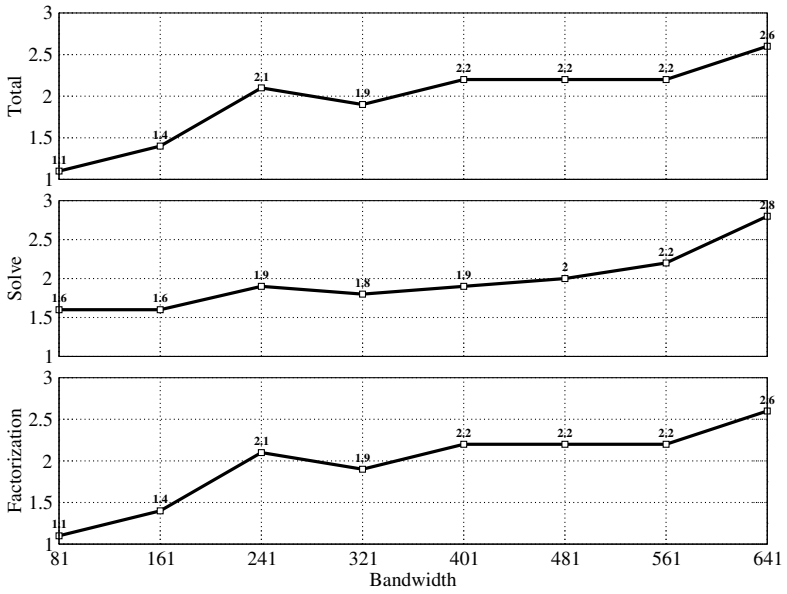


Figure 1.3: *Speed improvement for Test 1 of Recursive SPIKE, with partial pivoting, over ScaLapack for non-diagonally dominant systems.*

steadily as the bandwidth increases until it reaches 11.5 for $b = 641$. This represents a significant speed improvement for the SPIKE algorithm than those presented in Fig.1.3.

For the solve stage, Test 2a shows better results than Test 1, with λ reaching 3.5, while in Test 2b λ averages only 1.3. The results are shown after one-half BiCGstab iteration (one or two iterative refinements are also enough to get the desired convergence). Although the solve stage is slower for this case, due to the time taken by the matrix-vector multiplications in the outer iterative scheme, overall SPIKE is still faster than ScaLapack by factors ranging from 2.0 to 11.0.

Test 3a and 3b

These experiments are concerned with diagonally dominant systems solved using the Truncated SPIKE algorithm. In Test 3a, SPIKE is used as a direct solver using the LU-UL factorization strategy. In Test 3b, only LU is performed on the diagonal blocks and the top blocks W_j^t are approximated, thus SPIKE is used as a preconditioner for a suitable outer iteration. Figure 1.5 shows the speed improvements over ScaLapack obtained using these two schemes. For the factorization stage of Test 3a, λ increases with increasing bandwidth from 1.3

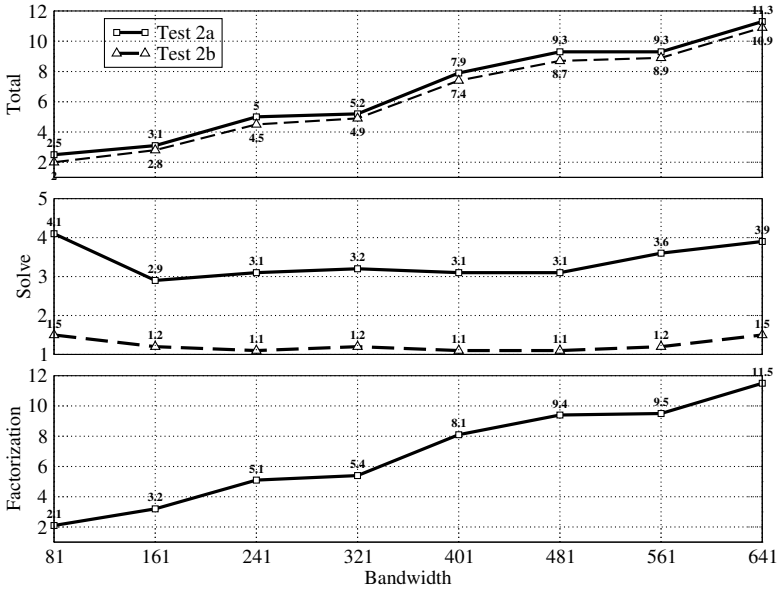


Figure 1.4: *Speed improvement over ScaLapack for Recursive SPIKE without pivoting for non-diagonally dominant matrices. No "zero-pivot" is detected in Test 2a, while in Test 2b outer-iterations are needed after diagonal boosting. In Test 2b, one half iteration of BiCGstab is necessary to satisfy the convergence criterion:* $r < 10^{-8}$

for $b = 81$ to 5.3 for $b = 641$. In Test 3b, the corresponding speed improvement roughly doubles indicating that the time taken to approximate the blocks W_j^t is negligible compared to the the time taken by the UL factorization. For the solve stage, the SPIKE and ScaLapack schemes are equivalent. For Test 3a, $\lambda = 1$, and for Test 3b, λ drops to roughly 0.2. This is due to the matrix-vector multiplications required by these 1 to 3 BiCGstab outer iterations needed for convergence. Overall, however, speed improvement over ScaLapack based on the total time is still significant as the bandwidth increases. We note that the version of SPIKE used in Test 3b yields higher speed improvement and thus could be of value in applications where the factorization and solve stages are not decoupled.

Additional remarks

Table 1.2 lists the different tests summarizing the time taken by both the factorization and solve stages of each algorithm for handling a given system with

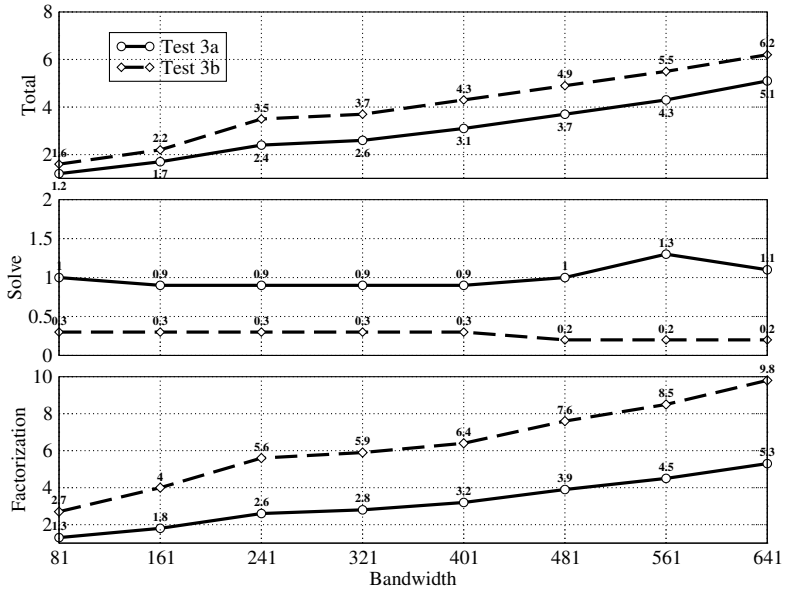


Figure 1.5: *Speed improvement over ScaLapack for the Truncated SPIKE algorithm for diagonally dominant systems. Test 3a uses the LU-UL strategy, while the Test 3b requires outer iterations because of the approximation used to compute W_j^t in the factorization stage.*

bandwidth $b = 401$. For the factorization stage, one can see that Test 1 concerned the most expensive SPIKE scheme – LU-factorization with partial pivoting. Without pivoting, but with a diagonal boosting strategy, this time drops significantly in Tests 2a and 2b. The LU-UL strategy used in Test 3a reduces the times consumed further by avoiding the generation of the spikes. This time is again cut in half in Test 3b by eliminating the need of the UL factorization. For the solve stage, the best time is obtained for Tests 2a and 3a as no outer iteration is involved, while outer iterations cause Tests 2b and 3b to be the most expensive.

1.6.2 Scalability

In this section, we consider a system of order $n = 1,920,000$ with a bandwidth $b = 401$. Different versions of SPIKE are used for diagonally or non-diagonally dominant systems on $p = 32$, 64, 128, 256 and 512 processors. We note that until $p = 512$, the size of each partition $n_j = n/p$ is much larger than $m = (b-1)/2$ (number of columns in each spike).

Table 1.2: *List of various tests associated with various options for the SPIKE algorithm augmented with factorization and solve times (in seconds) obtained for $b = 401$. ScaLapack times are also given for comparisons.*

	Diagonally Dominant	Factorization step	Solve step	Outer-iterations	Spike Time(s) Fact., Solve	Scal. Time(s) Fact., Solve
Test 1	No	LU with pivoting	Recursive	No	9.11, 0.16	20.41, 0.30
Test 2a	No	LU without pivoting + boost.	Recursive	No	2.56, 0.10	20.61 0.31
Test 2b	No	LU without pivoting + boost.	Recursive	Yes	2.55, 0.29	20.55 0.32
Test 3a	Yes	LU-UL without pivoting	Truncated	No	1.22, 0.09	3.92 0.082
Test 3b	Yes	LU without pivoting + approx.	Truncated	Yes	0.6, 0.32	3.87 0.082

Non-diagonally dominant systems

Two versions of SPIKE are considered: the Recursive SPIKE algorithm with pivoting used in Test 1, and the Recursive SPIKE without pivoting used in Test2b. Figures 1.6 and 1.7, respectively compare the times consumed for the factorization and the solve stages. For the factorization stage, both versions of SPIKE are faster than ScaLapack. Although SPIKE without pivoting is the fastest algorithm, its scalability is not as robust due to the communication costs on the IBM-SP. However, as the ratio n^j/m increases (up to 128), the computational time tends to be more dominant than that required for interprocessor communications resulting in good scalability on the IBM-SP. For the solve stage, both versions of Spike perform better than ScaLapack. In all the cases, for this computing platform, scalability deteriorates beyond 128 processors as the interprocessor communication time required for solving the reduced system increases.

Diagonally dominant systems

As in Test 3a, we consider the LU-UL strategy for the Truncated SPIKE. In Fig. 1.8, one can see that the factorization times of SPIKE are lower than those of ScaLapack, with reasonable scalability until 128 processors. The results obtained for the solve stage in Fig. 1.9 show that the ScaLapack and SPIKE times are equivalent until 256 processors and the scaling is almost optimal. However, for 512 processors, SPIKE proves to be more scalable than ScaLapack. Indeed, using the Truncated version of SPIKE, the communication time to solve the reduced system is minimal, and the computational time dominates that required by interprocessor communication.

Table 1.3: *Test 1- Simulation times and relative speed improvement (λ) vs ScaLapack, obtained with the Recursive SPIKE with partial pivoting for non-diagonally dominant systems.*

	Factorization			Solve			Total		
Bandwidth	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)
b=81	0.43	**1.1**	0.41	0.088	**1.6**	0.054	0.52	**1.1**	0.46
b=161	1.64	**1.4**	1.18	0.121	**1.6**	0.078	1.76	**1.4**	1.25
b=241	5.22	**2.1**	2.45	0.19	**1.9**	0.10	5.42	**2.1**	2.55
b=321	8.83	**2.2**	4.71	0.24	**1.8**	0.13	9.07	**1.9**	4.84
b=401	20.41	**2.2**	9.11	0.30	**1.9**	0.16	20.72	**2.2**	9.28
b=481	34.69	**2.2**	15.57	0.37	**2.0**	0.18	35.06	**2.2**	15.75
b=561	47.91	**2.2**	22.11	0.48	**2.2**	0.22	48.39	**2.2**	22.33
b=641	75.70	**2.6**	29.20	0.69	**2.8**	0.25	76.39	**2.6**	29.45

Table 1.4: *Test 2a- Simulation times and relative speed improvement (λ) w.r.t. ScaLapack, obtained with the Recursive SPIKE without pivoting where no-outer iteration is needed, for non-diagonally dominant systems.*

	Factorization			Solve			Total		
Bandwidth	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)
b=81	0.49	**2.4**	0.21	0.090	**4.1**	0.022	0.58	**2.5**	0.23
b=161	1.63	**3.1**	0.53	0.130	**2.9**	0.044	1.75	**3.1**	0.57
b=241	5.24	**5.1**	1.03	0.20	**3.1**	0.064	5.44	**5.0**	1.10
b=321	8.83	**5.3**	1.65	0.25	**3.2**	0.078	9.08	**5.2**	1.73
b=401	20.61	**8.1**	2.56	0.31	**3.1**	0.099	20.61	**7.9**	2.66
b=481	34.75	**9.5**	3.68	0.37	**3.1**	0.12	35.12	**9.3**	3.79
b=561	47.99	**9.5**	5.05	0.48	**3.6**	0.14	48.47	**9.3**	5.19
b=641	75.69	**11.5**	6.56	0.66	**3.9**	0.17	76.36	**11.3**	6.74

Table 1.5: *Test 2b- Simulation times and relative speed improvement (λ) w.r.t. ScaLapack, obtained with the Recursive SPIKE without pivoting where outer iterations are needed, for non-diagonally dominant systems.*

	Factorization			Solve			Total		
Bandwidth	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)
b=81	0.44	**2.1**	0.21	0.090	**1.5**	0.060	0.59	**2.0**	0.28
b=161	1.68	**3.2**	0.53	0.14	**1.2**	0.12	1.83	**2.8**	0.65
b=241	5.26	**5.1**	1.04	0.20	**1.1**	0.17	5.46	**4.5**	1.21
b=321	8.93	**5.4**	1.66	0.27	**1.2**	0.23	9.20	**4.9**	1.89
b=401	20.55	**8.1**	2.55	0.32	**1.1**	0.29	20.87	**7.4**	2.84
b=481	34.27	**9.4**	3.66	0.38	**1.1**	0.34	34.64	**8.7**	4.00
b=561	47.85	**9.5**	5.04	0.47	**1.2**	0.39	48.33	**8.9**	5.44
b=641	75.84	**11.5**	6.58	0.72	**1.5**	0.47	76.56	**10.9**	7.05

Table 1.6: *Test 3a- Simulation times and relative speed improvement (λ) w.r.t. ScaLapack, obtained with the Truncated SPIKE with LU-UL strategy, for diagonally dominant systems.*

	Factorization			Solve			Total		
Bandwidth	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)
b=81	0.20	**1.3**	0.16	0.022	**1.0**	0.022	0.22	**1.2**	0.18
b=161	0.63	**1.8**	0.35	0.041	**0.9**	0.044	0.67	**1.7**	0.40
b=241	1.58	**2.6**	0.61	0.056	**0.9**	0.062	1.64	**2.4**	0.67
b=321	2.38	**2.8**	0.85	0.068	**0.9**	0.073	2.45	**2.6**	0.93
b=401	3.92	**3.2**	1.22	0.082	**0.9**	0.090	4.00	**3.1**	1 .31
b=481	6.40	**3.9**	1.65	0.10	**1.0**	0.10	6.51	**3.7**	1.75
b=561	9.64	**4.5**	2.16	0.15	**1.3**	0.12	9.79	**4.3**	2.28
b=641	14.36	**5.3**	2.72	0.16	**1.1**	0.14	14.52	**5.1**	2.87

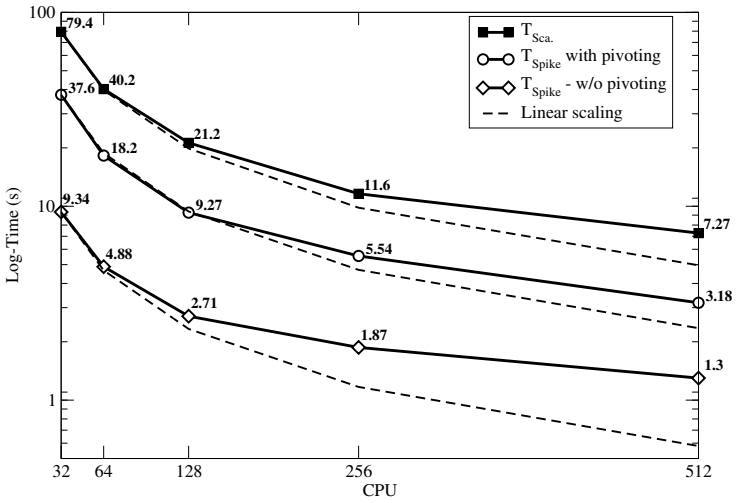


Figure 1.6: *Factorization times (on log scale) taken by ScaLapack, Recursive SPIKE with pivoting, and Recursive SPIKE without pivoting for non-diagonally dominant systems.*

Table 1.7: *Test 3b- Simulation times and relative speed improvement (λ) w.r.t. ScaLapack, obtained with the Truncated SPIKE with LU factorization and an approximation strategy, for diagonally dominant systems.*

	Factorization			Solve			Total		
Bandwidth	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)	$T_{Sca.}$(s)	λ	T_{Spike}(s)
b=81	0.20	**2.7**	0.076	0.022	**0.3**	0.062	0.22	**1.6**	0.14
b=161	0.68	**4.0**	0.17	0.042	**0.3**	0.016	0.72	**2.2**	0.33
b=241	1.58	**5.6**	0.28	0.056	**0.3**	0.18	1.64	**3.5**	0.46
b=321	2.47	**5.9**	0.42	0.072	**0.3**	0.26	2.54	**3.7**	0.68
b=401	3.87	**6.4**	0.60	0.082	**0.3**	0.32	3.95	**4.3**	0.92
b=481	6.42	**7.6**	0.85	0.10	**0.2**	0.47	6.53	**4.9**	1.32
b=561	9.70	**8.5**	1.15	0.15	**0.2**	0.66	9.85	**5.5**	1.80
b=641	14.45	**9.8**	1.48	0.17	**0.2**	0.89	14.61	**6.2**	2.37

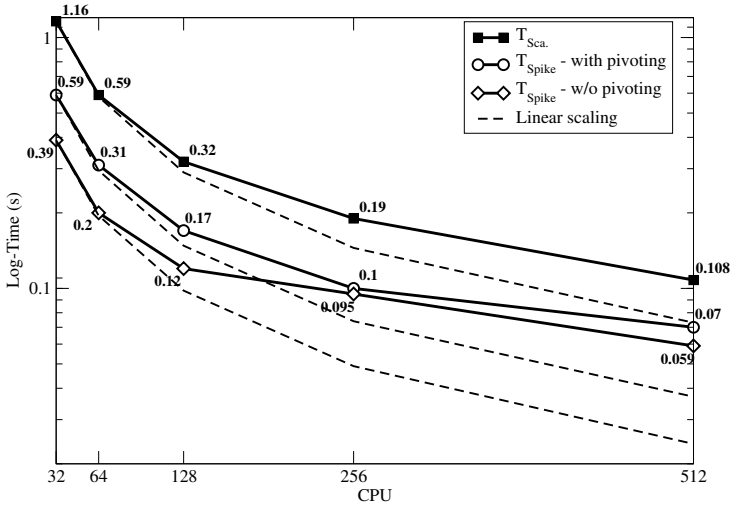


Figure 1.7: *Solve times (on log scale) taken by ScaLapack, Recursive SPIKE with pivoting, and Recursive SPIKE without pivoting for non-diagonally dominant systems.*

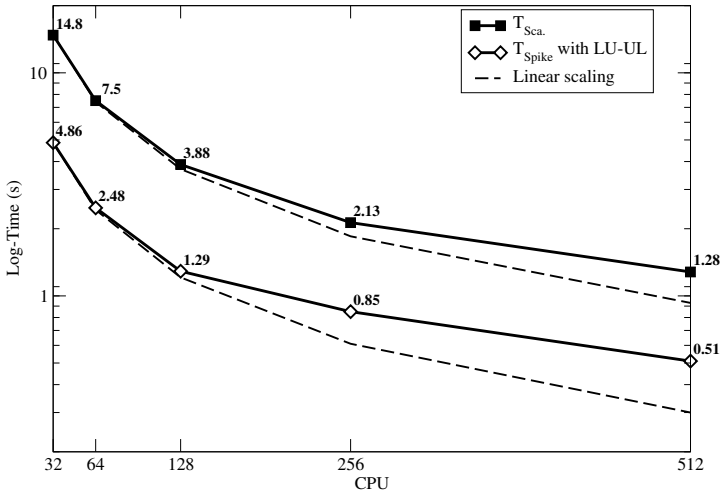


Figure 1.8: *Time for the factorization stage (on log scale) for ScaLapack, and the Truncated SPIKE using LU-UL strategy, for diagonally dominant systems.*

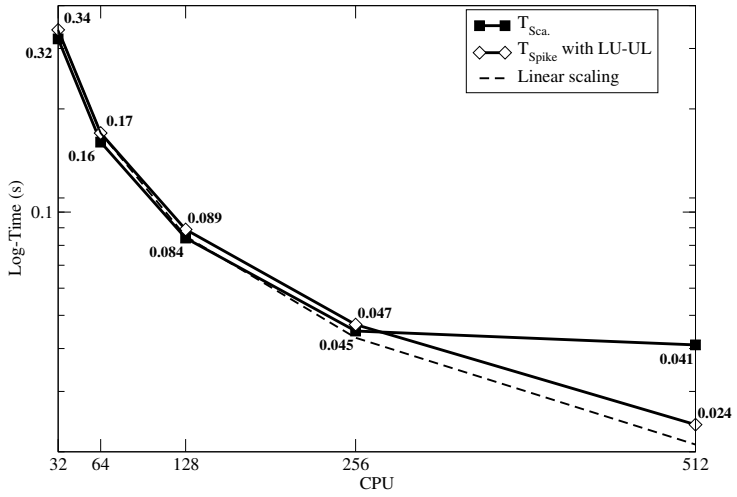


Figure 1.9: *Time for the solve stage (on log scale) for ScaLapack, and the Truncated SPIKE using LU-UL strategy, for diagonally dominant systems.*

Chapter 2

A Parallel Framework for Solving Banded Linear Systems that are Sparse Within the Band

2.1 Introduction

A large number of applications produce narrow banded systems (with or without reordering) that are sparse within the band. For these systems in which the bandwidth is not sufficiently narrow so as to be treated as dense within the band, we need to develop an alternate version of SPIKE. A SPIKE "on-the-fly" scheme is proposed that does not require the generation of the spikes, and for which the reduced system is not formed explicitly. Rather, the reduced system is solved via a matrix-free iterative scheme (such as BiCGstab) in which the matrix-vector multiplications are performed "on-the-fly". This scheme is ideally suited for handling banded systems with large sparse bands. In addition, a multi-level parallel version of SPIKE is proposed to take advantage of parallel sparse direct solvers such as SupeLU [52], MUMP'S [56] or PARDISO [58].

Solving the reduced system implicitly

The top and bottom parts of the spikes V_j and W_j that constitute the reduced system, can be expressed as follows:

$$V_j^{(b)} = \begin{bmatrix} 0 & I_m \end{bmatrix} (A_j)^{-1} \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j, \quad W_j^{(t)} = \begin{bmatrix} I_m & 0 \end{bmatrix} (A_j)^{-1} \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j,$$

$$V_j^{(t)} = \begin{bmatrix} I_m & 0 \end{bmatrix} (A_j)^{-1} \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j, \quad W_j^{(b)} = \begin{bmatrix} 0 & I_m \end{bmatrix} (A_j)^{-1} \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j.$$

2.2 A sparse SPIKE system solver using multi-level parallelism

For banded systems that are sparse within the band, existing parallel sparse direct solvers such as SupeLU, MUMPS, or PARDISO, may not be successful in minimizing the fill-in in the factorization stage even after reordering. This is particularly true if the matrix is narrow banded, i.e., the ratio of the bandwidth to the system size is small (ranging from 0.001 to 0.01). In the SPIKE scheme, the partitioning stage will create diagonal blocks with larger bandwidth-size ratios that allow efficient use of these sparse direct solvers on each partition. To minimize the time taken by the solve stage of the SPIKE "on-the-fly" scheme, we employ a two-level parallel scheme for obtaining a reduced system of much smaller size, and thus easier to solve by an iterative scheme. For instance in Fig. 2.1, we consider four partitions and make use of the PARDISO shared memory direct solver in each partition (i.e., node), while the SPIKE distributed memory scheme is used across partitions (i.e., nodes).

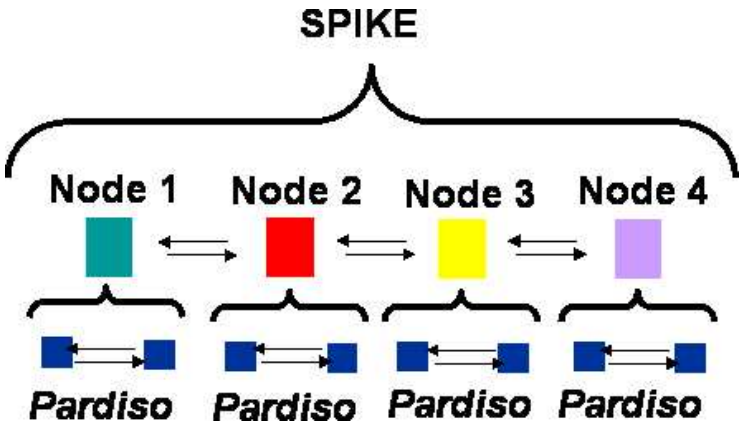


Figure 2.1: *SPIKE with two-level parallelism using PARDISO within each node. The machine used for the illustration, has 4 dual-processor nodes.*

Table 2.1: *Test (a). SPIKE runs using PARDISO on 2 and 4 nodes and Truncated SPIKE on 8 processors, with Reordering, Factorization, Solve, and Total times in seconds.*

Test (a)	Reord.	Fact.	Solve	Total	Residual
SPIKE 2-nodes	9.48	1.55	4.54	15.57	10^{-7}
SPIKE 4-nodes	5.68	0.77	2.3	8.76	10^{-7}
Trunc. SPIKE (8)	0	0.84	0.14	0.98	10^{-14}

2.3 Numerical experiments

We consider two test cases for non-diagonally dominant symmetric positive definite systems with different bandwidths. The systems are obtained via a 3-D finite element discretization of the Poisson equation [59]:

Test (a) size $n = 432,000$, bandwidth $b = 177$, number of non-zero elements $nnz = 7,955,116$, with 10.4% sparsity density within the band.

Test (b) $n = 471,800$, $b = 1455$, $nnz = 9,499,744$, 1.4% sparsity density within the band.

Our numerical experiments are performed on an Intel dual-Xeon 3.2 Ghz Linux cluster of four nodes with Infiniband connection. Observing that the PARDISO package is included in the Intel-MKL library, Table 2.1 shows the time consumed in the various stages of SPIKE with PARDISO on 2 and 4 nodes, and the Truncated SPIKE on 8 processors assuming that the matrix is dense within the band.

Table 2.2: *Test (b). SPIKE runs using PARDISO on 2 and 4 nodes, with Reordering, Factorization, Solve, and Total times in seconds.*

Test (b)	Reord.	Fact.	Solve	Total	Residual
SPIKE 2-nodes	18.69	11.06	23.06	52.82	10^{-7}
SPIKE 4-nodes	8.41	6.54	10.85	25.81	10^{-7}

One observes that for such narrow banded systems as proposed in test (a), SPIKE becomes much more effective (speed and accuracy) if we consider the matrix as dense within the band. Also, as stated previously, we demonstrate that the Truncated SPIKE scheme converges nicely even if the system in test (a) is not diagonally dominant. The SPIKE-PARDISO "on-the-fly" scheme shows good scalability for the reordering, factorization and solve stages, where the outer iterations are performed using BiCGstab with a relative residual stopping criterion of 10^{-7}. In contrast, if one uses PARDISO alone on one and two processors on the same system, the reordering times remain almost unchanged at $18.5s$, while the factorization times range from 4.1s to 3.1s. Also, the times consumed by the distributed memory solvers MUMPS and SuperLU on 1, 2, 4 and 8 processors associated with 1, 2, 4, and 4 nodes, respectively, are given by:

- reordering stage: $\sim$ 17s for MUMPS (for all parallel configurations), and from 10.5s to 7.5s for SuperLU,
- factorization stage: 6.3s, 4.2s, 3s and 20s for MUMPS with excessive memory swaps on 8 processors, and from 17s to 180s for SuperLU due to too much fill-in and memory swaps on our machine (with 4GB of memory per node).

In contrast to test (a), the system in test (b) cannot be treated as dense within the band. Table 2.2 shows the results realized by SPIKE with PARDISO on 2 and 4 nodes. SPIKE-PARDISO "on-the-fly" also shows very good scalability, with more than a factor two for the reordering stage. Due to excessive fill-in in the factorization stage, both MUMPS and SuperLU require more memory than available in our platforms.

Chapter 3

Weighted Matrix Ordering and Banded Preconditioners for Non-symmetric Linear System Solvers

3.1 Introduction

Solving sparse linear systems is the most time consuming part of many applications in computational science and engineering. While direct methods provide robust solvers (*i.e.*, they are guaranteed to find an existing solution in a precisely characterizable amount of time), their application is feasible only if the system is not very large. Iterative methods, on the other hand, take advantage of system properties to provide good approximations to the solution in much shorter times. Furthermore, while iterative solvers are more scalable on parallel architectures, they are not as robust as direct solvers. In iterative solvers, preconditioners are often used to improve the convergence properties. Unlike approximate factorization-based preconditioners, banded preconditioners have excellent parallel scalability [61].

In order to apply a banded solver, it is necessary to reduce the bandwidth of the system. While traditional bandwidth reduction techniques such as Reverse Cuthill-McKee [20] and Sloan [62], are successful for classes of problems, their application for extraction of effective banded preconditioners is limited. This is because, the performance of these algorithms depends on the intrinsic sparsity pattern of the matrix under consideration, *i.e.*, they are only applicable if the matrix can be reordered into a narrow banded system, so that a direct solver can

be used. Parallel banded solvers, such as SPIKE [67], also require a reasonably small bandwidth, since the volume of communication is directly dependent on the bandwidth. Clearly, this may not be the case for most linear systems. Symmetric reordering alone does not guarantee a nonsingular preconditioner. In fact, in many applications such as circuit or chemical process simulation, the main diagonal is likely to contain many zeros raising the likelihood of a singular banded preconditioner.

To overcome these problems, we developed a bandwidth reduction technique aimed at encapsulating as many of the heaviest elements of the matrix into a central narrow band. This makes it possible to extract a narrow-banded preconditioner by dropping the entries that are outside the band. To solve the *weighted bandwidth reduction* problem, we use a *weighted spectral reordering* (WSO) technique that provides an optimal solution to a continuous relaxation of the corresponding optimization problem. This technique is a generalization of spectral reordering, which has also been effectively used for reducing the bandwidth and envelope of sparse matrices [4]. To alleviate the problems associated with symmetric reordering, we couple this method with non-symmetric reordering techniques, such as the maximum traversal algorithm [25], to make the main diagonal zero free and place the largest entries on the main diagonal [26].

The resulting algorithm can be summarized as follows: (i) use non-symmetric reordering to make the main diagonal free of zeros, (ii) use weighted spectral reordering to move larger elements closer to the diagonal, and (iii) extract a narrow central band to use as a preconditioner for a Krylov subspace method. Our results show that this yields a fast, highly parallelizable preconditioner with excellent convergence characteristics, particularly for non-symmetric matrices with significantly variable entries in terms of their magnitude. We also demonstrate that WSO is more robust than LU-type preconditioners.

3.2 Background and Related Work

Solving sparse linear systems,

$$Ax = f, \tag{3.1}$$

is the most time consuming part of many applications in scientific computing. Direct solvers, although very robust, are feasible if the system is not very large. Iterative methods, on the other hand, are more suitable for very large sparse systems, but lack robustness.

Preconditioning aims to improve the robustness of iterative methods by transforming the system into

$$M^{-1}Ax = M^{-1}f, \text{ or } AM^{-1}(Mx) = f. \tag{3.2}$$

Here, the preconditioner M is designed in such a way that the coefficient matrix AM^{-1} or $M^{-1}A$ has more desirable properties. In this report, without loss of generality, we use the term *preconditioning* to refer to left preconditioning. Here, M^{-1} is an approximation to A^{-1} in the sense that the eigenvalues of $M^{-1}A$ are

clustered around 1. In addition, the action of M^{-1} on a vector should be cheap and suitable for parallel computing.

An important class of preconditioners is based on approximate LU-factorization. These ILU preconditioners are well studied and used successfully to precondition various classes of linear systems [17, 35, 82]. There also exist dedicated software packages that are commonly used, including the ILUT package [63]. More recently, Benzi et al. investigated the role of reordering on the performance of ILU preconditioners [6, 7]. Banded preconditioners provide an effective alternative to ILU-type preconditioners because of their suitability for implementation on parallel architectures.

3.2.1 Banded preconditioners

A preconditioner M is banded if

$$kl \geq i - j \quad and \quad ku \geq j - i \Rightarrow m_{ij} \neq 0, \tag{3.3}$$

where kl and ku are the lower and upper bandwidths, respectively. The half-bandwidth, k, is defined as the maximum of kl and ku, and the bandwidth of the matrix is equal to $kl + ku + 1$. Banded preconditioners are generalizations of diagonal and tridiagonal preconditioners and are shown to be effective in solving various problems [57, 61, 77]. However, banded preconditioners alone are not suitable as black-box preconditioners, since a dominant band does not always exist without preprocessing. Therefore, it is necessary to reorder the rows and columns of a matrix to pack them in a narrow central band that can be used as a preconditioner. In general, reordering algorithms solve an optimization problem that can be defined as one of finding a permutation of the rows and columns that minimizes the bandwidth. Our approach is based on a generalization of this approach in that we aim to minimize the bandwidth that encapsulates a portion of the non-zeros in the matrix, rather than all non-zeros.

3.3 Weighted Bandwidth Reduction

3.3.1 Non-symmetric reordering

We first apply a non-symmetric row permutation of (3.1) as follows:

$$QAx = Qf. \tag{3.4}$$

Here, Q is a permutation matrix. We consider two cases: (i) find a Q such that the main diagonal contains as many non-zeros as possible, (ii) find a Q such that the product of the absolute values of the diagonal entries is maximized [26]. Optionally, the second approach can provide scaling factors so that the absolute values of the diagonal entries are equal to one and all other elements are less than or equal to one. Scaling is applied to the original coefficient matrix as follows:

$$(QD_2AD_1)(D_1^{-1}x) = (QD_2f). \tag{3.5}$$

The first algorithm is known as *maximum traversal search*. Both algorithms are implemented in the MC64[25] subroutine of the Harwell Subroutine Library HSL[45].

3.3.2 Symmetric reordering

After the above non-symmetric reordering and optional scaling, we apply a symmetric reordering as follows:

$$(PQD_2AD_1P^T)(PD_1^{-1}x) = (PQD_2f). \tag{3.6}$$

where P is a permutation matrix. We use Weighted Spectral Reordering (WSO) to find a permutation that minimizes the bandwidth encapsulating a specified fraction of the total magnitude of non-zeros in the matrix. "WSO" is described in detail in the next section.

Weighted spectral reordering (WSO)

Traditional reordering algorithms, such as Cuthill-McKee [20] and spectral reordering [4], aim to minimize the bandwidth of a matrix. The half-bandwidth of a matrix A is defined as

$$BW(A) = \max_{i,j:A(i,j)>0} |i-j|, \tag{3.7}$$

i.e., the maximum distance of a nonzero entry from the main diagonal. These methods aim to pack the non-zeros of a sparse matrix into a narrow band around the diagonal.

These methods do not take into account the magnitude of nonzero entries. However, significantly degrade the performance (*e.g.*, convergence rate) of the algorithm at hand, particularly for matrices whose entries vary significantly in magnitude.

To address this problem, we introduce the weighted bandwidth reduction problem, which aims at packing the heaviest elements of the matrix in a narrow band. This results in a less constrained formulation of the bandwidth reduction problem. In other words, the goal of weighted bandwidth reduction is to obtain a preconditioner with minimal bandwidth, which adequately *approximates* the matrix at hand, rather than minimizing the bandwidth of the entire matrix. More precisely, for matrix A, and specified error bound ϵ, we define the weighted bandwidth reduction problem as one of finding a matrix M with minimum bandwidth, such that

$$\frac{\sum_{i,j}|A(i,j)| - \sum_{i,j}|M(i,j)|}{\sum_{i,j}|A(i,j)|} \leq \epsilon. \tag{3.8}$$

The idea behind this formulation is that, if a significant part of the matrix is packed in a narrow band, then the rest of the non-zeros can be dropped to obtain

an efficient preconditioner, while keeping the effect of the resulting perturbation within a specified bound.

In order to find a heuristic solution to the weighted bandwidth reduction problem, we use a generalization of spectral reordering. Spectral reordering is a linear algebraic optimization technique that is commonly used to obtain approximate solutions to various intractable graph optimization problems [44]. It is also successfully applied to the bandwidth and envelope reduction problems for sparse matrices [4]. The core idea of spectral reordering is to compute a vector x that minimizes

$$\sigma_A(x) = \sum_{i,j:A(i,j)>0} (x(i) - x(j))^2, \tag{3.9}$$

where $||x||_2 = 1$. Here, it is assumed that the matrix A is symmetric. The vector x that minimizes $\sigma_A(x)$ provides a mapping of the rows (and columns) of the matrix A to a one-dimensional Euclidean space, such that pairs of rows that correspond to non-zeros are located as close as possible to each other. Consequently, the ordering of the entries of the vector x provides an ordering of the matrix that significantly reduces the bandwidth.

Fiedler [29] showed that the optimal solution to this problem is given by the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix, where the Laplacian L of a matrix A is defined as

$$\begin{array}{ll} L(i,j) = -1 & \textbf{if } i \neq j \wedge A(i,j) > 0 \\ L(i,i) = |\{j : A(i,j) > 0\}|. & \end{array} \tag{3.10}$$

Note that the matrix L is positive semi-definite, so the smallest eigenvalue of this matrix is equal to zero. The eigenvector x that minimizes $\sigma_A(x) = x^T L x$ is known as the Fiedler vector. The Fiedler vector of a sparse matrix can be computed efficiently using iterative algorithms [49].

While spectral reordering is shown to be effective in bandwidth reduction, the classical spectral approach described above ignores the magnitude of non-zeros in the matrix. Therefore, it is not directly applicable to the weighted bandwidth reduction problem. However, Fiedler's result can be generalized for the weighted case [14]. More precisely, the eigenvector x that corresponds to the second smallest eigenvalue of the weighted Laplacian $\bar{L}$ minimizes

$$\bar{\sigma}_A(x) = x^T \bar{L} x = \sum_{i,j} |A(i,j)|(x(i) - x(j))^2, \tag{3.11}$$

where $\bar{L}$ is defined as

$$\begin{array}{ll} \bar{L}(i,j) = -|A(i,j)| & \textbf{if } i \neq j \\ \bar{L}(i,i) = \sum_j |A(i,j)|. & \end{array} \tag{3.12}$$

We now show how weighted spectral reordering can be used to obtain a continuous approximation to the weighted bandwidth reduction problem. For

this purpose, we first define the relative band-weight of a given band in the matrix as follows:

$$w_k(A) = \frac{\sum_{i,j:|i-j|<k} |A(i,j)|}{\sum_{i,j} |A(i,j)|}. \tag{3.13}$$

In other words, the band-weight of a matrix A, with respect to an integer k, is equal to the fraction of the total magnitude of entries that are encapsulated in a band of half-width k.

For a given α, $0 \leq \alpha \leq 1$, we define α-bandwidth as the smallest half-bandwidth that encapsulates a fraction α of the total matrix weight, *i.e.*,

$$BW_\alpha(A) = \min_{k:w_k(A)\geq\alpha} k. \tag{3.14}$$

Observe that α-bandwidth is a generalization of half-bandwidth, *i.e.*, when $\alpha = 1$, the α-bandwidth is equal to the half-bandwidth of the matrix.

Now, for a given vector $x \in \mathbb{R}^n$, define an injective permutation function $\pi(i) : \{1, 2, ..., n\} \to \{1, 2, ..., n\}$, such that, for $1 \leq i, j \leq n$, $x(\pi(i)) \leq x(\pi(j))$ iff $i \leq j$. Here, n denotes the number of rows (columns) in matrix A. Moreover, for fixed k, define the function $\delta_k(i, j) : \{1, 2, ..., n\} \times \{1, 2, ..., n\} \to \{0, 1\}$, which quantizes the difference between $\pi(i)$ and $\pi(j)$ with respect to k, *i.e.*,,

$$\delta_k(i,j) = \begin{cases} 0 & \text{if } |\pi(i) - \pi(j)| \leq k \\ 1 & \text{else} \end{cases} \tag{3.15}$$

Let $\bar{A}$ be the matrix obtained by reordering the rows and columns of A according to π, *i.e.*,

$$\bar{A}(\pi(i), \pi(j)) = A(i,j) \text{ for } 1 \leq i, j \leq n. \tag{3.16}$$

Then, $\delta_k(i, j) = 0$ indicates that $A(i, j)$ is inside a band of half-width k in matrix A, while $\delta_k(i, j) = 1$ indicates that it is outside this band. Defining

$$\hat{\sigma}_k(A) = \sum_{i,j} |A(i,j)|\delta_k(i,j), \tag{3.17}$$

we obtain

$$\hat{\sigma}_k(A) = (1 - w_k(\bar{A})) \sum_{i,j} |A(i,j)|. \tag{3.18}$$

Therefore, for fixed α, the α-bandwidth of matrix $\bar{A}$ is equal to the smallest k that satisfies $\hat{\sigma}_A(k)/\sum_{i,j} |A(i,j)| \leq 1 - \alpha$.

Observe that the problem of minimizing $\bar{\sigma}_x(A)$ is a continuous relaxation to the problem of minimizing $\hat{\sigma}_k(A)$ for given k. Therefore, the Fiedler vector of weighted Laplacian $\bar{L}$ provides a good basis for reordering A to minimize $\hat{\sigma}_k(A)$. Consequently, for fixed ϵ, this vector provides a heuristic solution to the problem of finding a reordered matrix $\bar{A}$ with minimum $(1 - \epsilon)$-bandwidth. Once this matrix is obtained, we compute the banded preconditioner M as follows:

$$M = \{M(i,j) : M(i,j) = \bar{A}(i,j) \text{ if } |i-j| \leq BW_{1-\epsilon}(\bar{A}),\ 0 \text{ else}\}. \tag{3.19}$$

Clearly, M satisfies Equation 3.8 and it is of minimal bandwidth.

Note that spectral reordering is defined specifically for symmetric matrices and the resulting permutation is symmetric as well. Since the main focus of this study is on non-symmetric matrices, we generalize spectral reordering to non-symmetric matrices by computing the Laplacian matrix with respect to $|A| + |A^T|$ instead of $|A|$. Note that, this formulation results in a symmetric permutation for a non-symmetric matrix, which may be considered over-constrained.

3.3.3 Summary of banded solvers

On parallel platforms, the Spike banded solver [10, 15, 23, 51, 67, 71] is shown to have excellent scalability [59, 60] compared to ScaLapack [11]. The central idea of SPIKE is to partition the matrix so that each processor can work on its own part and communicates with other processors only at the end to solve a common reduced system. The size of the reduced system is determined by the bandwidth of the matrix and the number of partitions.

3.4 Numerical Experiments

With the objective of establishing superior runtime and convergence properties of banded preconditioners for a broad class of problems, we perform detailed experimental measurements.

3.4.1 Experimental setup and test problems

We obtained the test problems from the University of Florida Sparse Matrix Collection [21]. We choose moderately large matrices with a larger number of non-zeros (to ensure that the bandwidth reduction problem is not trivial). We did not include those matrices that can be reordered into a narrow band using classical reordering schemes such as Reverse Cuthill-McKee (RCM). Further, in order to demonstrate the effectiveness of our methods, we chose matrices from various applications that are structurally un-symmetric and have zeros on the main diagonal. Therefore, without any reordering, they are difficult to solve using banded or ILU type preconditioners. Description of the test problems and related statistics are shown in Tables 3.1 and 3.2, respectively.

In all of the numerical experiments, we use the BiCGStab [78] iterative solver with a left preconditioner. We terminate the iterations when $||r_k||_\infty / ||r_0||_\infty \leq 10^{-5}$. The right hand side is generated from a solution vector of all ones in order to ensure that $f \in span(A)$.

Implementation of ILU based preconditioners For each problem, we apply ILUT to precondition the system as follows. For non-symmetric permutation, we implement the most promising technique mentioned in [6, 7]. Namely,

Table 3.1: *Description of Test Problems*

Number	Name	Application
1	2D_54019_HIGHK	Semiconductor Device Simulation
2	APPU	NASA App benchmark
3	ASIC_680k	Circuit Simulation Problem
4	BUNDLE1	3D Computer Vision
5	DC1	Circuit Simulation Problem
6	DW8191	Dielectric Waveguide
7	FEM_3D_THERMAL1	Thermal Problem
8	FINAN512	Economic Problem
9	FP	2-D Fokker Planck Equations
10	H2O	Theoretical/Quantum Chemistry Problem
11	MSC23052	Symmetric Test Matrix from MSC/NASTRAN
12	RAEFSKY5	Landing Hydrofoil Airplane FSE Model

Table 3.2: *Properties of Test Properties*

Number	Name	Dimension	Non-zeros
1	2D_54019_HIGHK	54,019	996,414
2	APPU	14,000	1,853,104
3	ASIC_680k	680,000	2,638,997
4	BUNDLE1	10,581	770,811
5	DC1	116,835	766,396
6	DW8191	8,192	41,746
7	FEM_3D_THERMAL1	17,880	430,740
8	FINAN512	74,752	596,992
9	FP	7,548	834,222
10	H2O	67,024	2,216,736
11	MSC23052	23,052	1,142,686
12	RAEFSKY4	19,779	674,195

we use the non-symmetric reordering algorithm available in MC64, with the objective of maximizing the absolute value of the product of diagonal entries. In addition, we obtain scaling factors (D_1 and D_2). After non-symmetric reordering and scaling, we perform RCM reordering. Finally, we obtain the ILUT factorization using the scaled and reordered coefficient matrix. We refer to this method as ILUTI. For ILUTI, we report the MC64 time, RCM time, ILUT factorization time, and the solution time, which involves the BiCGStab iterations.

We also use the ILUT preconditioner without any reordering or scaling, and we refer to this approach as ILUT. For both ILUTI and ILUT, we try various maximum fill-in ($fillin$) and drop ($droptol$) tolerance values. First, we allow a maximum $fillin$ of $10,000$ per row for all problems, and a $droptol$ of 10^{-1} and 10^{-3}. Note that, even though the maximum $fillin$ per row is quite generous, the actual $fillin$, see Table 3.4, due to dropping never reaches the level of the specified maximum $fillin$. In addition, as a separate experiment, we allow $fillin = k$, where k is the $(1-\epsilon)$-bandwidth of WSO, for a specified bound on error ϵ. In this case, we do not drop any elements. By considering various combinations of the $(fillin, droptol)$ parameter pair, one can experiment to find the optimal values. However, it turns out to be quite expensive to form the factorization each time and the behavior of performance with respect to varying these parameters is sometimes counter-intuitive. In other words, more $fillin$ does not always mean improving the convergence properties as previously reported [48, 64].

Implementation of weighted bandwidth reduction based preconditioner For each problem, we first reorder the system with MC64 using the variation of the algorithm that maximizes the number of non-zeros on the main diagonal. Next, we reorder the resulting system with spectral reordering using MC73[46], which is available in HSL. We use the default parameters of MC73. After obtaining the reordered system, we determine the $(1-\epsilon)$-bandwidth, where ϵ specifies the desired bound on the relative difference between the preconditioner and the original matrix. Note that the computation of the $(1-\epsilon)$-bandwidth requires $O(nnz) + O(n)$ time and $O(n)$ storage. To achieve this, we first create a work array, $w(1:n)$, of dimension n. Then, for each nonzero, a_{ij}, we update $w(abs(i-j)) = w(abs(i-j)) + abs(a_{ij})$. Finally, we compute a cumulative sum in w starting from index 0 until $(1-\epsilon)$-bandwidth is reached. The time for this process is included in the LAPACK time. In this set of experiments, we choose $\epsilon = 10^{-4}$. In general, the corresponding $(1-\epsilon)$-bandwidth can be as large as n. To prevent this, we place an upper limit $k \leq 50$ if the matrix dimension is greater than $10,000$. In Table 3.3, the achieved values of the $(1-\epsilon)$-bandwidth are given for each of the test problems.

After determining k, the WSO scheme proceeds with the extraction of the banded preconditioner with the bandwidth $2 \times k + 1$. Again, this process is inexpensive and its cost is reported together with the LAPACK time. After factorization of the preconditioner via an LU-scheme without pivoting, we use it to accelerate the BiCGStab iterations.

Solution of linear systems For both ILUTI and WSO, we obtain a solution to (3.6) by solving the following reordered and scaled (if any) system

$$\tilde{A}\tilde{x} = \tilde{f}. \tag{3.20}$$

Here, $\tilde{A} = PQD_2AD_1P^T$, $\tilde{x} = PD_1^{-1}x$, and $\tilde{f} = PQD_2f$. The corresponding scaled residual is given by $\tilde{r} = \tilde{f} - \tilde{A}\tilde{x}$, where $\tilde{x}$ is the computed solution. One can recover the residual of the original system (3.1), r, as follows:

$$\tilde{r} = PQD_2f - PQD_2AD_1P^TPD_1^{-1}x \tag{3.21}$$
$$\Rightarrow \tilde{r} = PQD_2f - PQD_2Ax \tag{3.22}$$
$$\Rightarrow \tilde{r} = PQD_2(f - Ax) \tag{3.23}$$
$$\Rightarrow \tilde{r} = PQD_2r \tag{3.24}$$
$$\Rightarrow r = D_2^{-1}Q^TP^T\tilde{r}. \tag{3.25}$$

At each iteration, we check the unscaled relative residuals using (3.25). Note that a permutation of the inverted scaling matrix is computed once and used for recovering the unscaled residual at each iteration. This process is included in the total time, but the cost is minimal compared to sparse matvecs and triangular solves. During our tests, we enforce a time limit of 10 minutes, and consider the particular run as failure if it exceeds this limit.

Table 3.3: *Values of fillin f and bandwidth k for the test problems*

Number	Name	fmax($fillin$)	k(α-bandwidth)
1	2D_54019_HIGHK	10,000	2
2	APPU	10,000	50
3	ASIC_680k	10,000	2
4	BUNDLE1	10,000	50
5	DC1	10,000	50
6	DW8191	10,000	190
7	FEM_3D_THERMAL1	10,000	50
8	FINAN512	10,000	50
9	FP	10,000	2
10	H2O	10,000	50
11	MSC23052	10,000	50
12	RAEFSKY4	10,000	50

3.4.2 Comparative analysis

In Figure 3.1, we compare the normalized time (with respect to the banded preconditioner) of ILUT and ILUTI using $droptol = 10^{-1}$ and $fillin = f_{max}$. ILUT fails in four of the test cases and diverges in one case, while ILUT does not fail but diverges for three of the test problems. WSO, on the other hand,

Table 3.4: *Number of nonzeros in matrix L + U, for various ILUT preconditioners*

Matrix	ILUT $(10^{-1}, fmax)$	ILUTI $(10^{-1}, fmax)$	ILUT $(10^{-3}, fmax)$	ILUTI $(10^{-3}, fmax)$	ILUT $(0, k)$	ILUTI $(0, k)$
1	-	419,835	-	1,503,034	1,213,809	1,217,347
2	2,377,197	2,310,466	-	-	-	-
3	-	2,473,454	-	2,783,146	-	4,457,554
4	347,395	290,337	10,247,157	1,368,809	1,815,975	992,708
5	607,692	707,162	1,230,897	1,100,108	-	-
6	43,702	54,189	476,802	1,239,229	1,577,605	1,626,048
7	247,643	256,741	762,011	695,664	2,225,895	2,217,004
8	472,830	478,568	2,336,798	1,427,693	5,191,052	5,059,937
9	-	2,875,303	2,104,578	4,301,556	892,662	892,129
10	1,306,190	1,249,849	7,602,509	7,316,812	-	-
11	-	26,402,760	4,515,416	27,939,415	2,417,059	3,294,876
12	923,285	858,737	5,166,173	3,911,335	3,194,353	3,296,441

succeeds in 11 of the 12 test cases. For the other test problem, namely the matrix $MSC23052$, WSO reaches a relative residual of 1.6×10^{-2}, while ILUT and ILUTI either diverge or fail. WSO is the fastest method based on the total solution time for 7 out of the 12 cases.

In Figure 3.2, we compare the normalized time (with respect to the banded preconditioner) of ILUT and ILUTI using $droptol = 10^{-3}$ and $fillin = f_{max}$. In this case, ILUT fails or diverges for four of the test cases, and it exceeds the time limit for one test case. ILUTI, on the other hand, diverges for three cases and stagnates for one. WSO is the the fastest method based on the total solution time for 9 out of the 12 cases.

In Figure 3.3, we compare the normalized time (with respect to the banded preconditioner) of ILUT and ILUTI using $droptol = 0$ and $fillin = k$. Here, $k = kl = ku$ is the upper (or lower) bandwidth of the banded preconditioner. ILUT diverges in three of the test cases, exceeds the time limit for three others, and fails for one. ILUTI, on the other hand, diverges for two of the test cases and exceeds the time limit for three others. WSO is the fastest method based on the total solve time.

In Table 3.5, we present details of the experimental results. Notice that there is no failure for WSO, whereas, ILUTI and ILUT have failures. If the ILUT factorization returns with an error code, we indicated this by an F in the table. If BiCGStab stagnates (indicated by *Stag*, or $*$), then we consider that the method did not converge to the desired relative residual of 10^{-5} after 500 iterations.

In Figures 3.4-3.15, we present the residual histories for 12 systems using ILUT, ILUTI, and WSO. The numbers of iterations for these methods are comparable. However, WSO is faster due to higher FLOP counts and better concurrency.

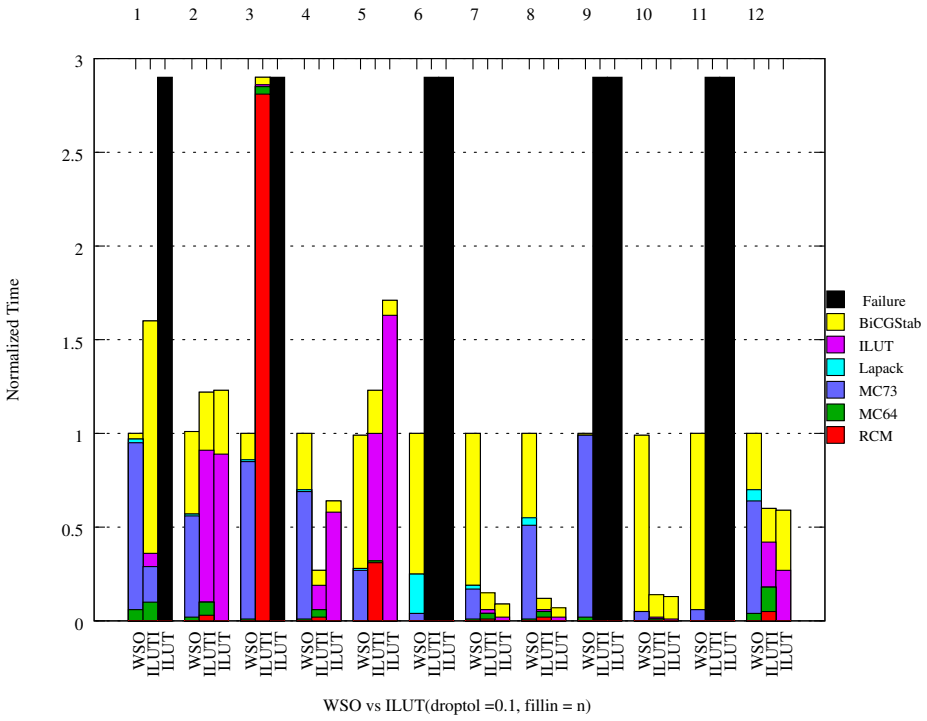


Figure 3.1: *Comparison of performance of WSO banded preconditioner with ILUT($f_{max}, 10^{-1}$)*

Table 3.5: *Comparison of ILUT,ILUTI, and WSO: Number of Iterations(Total Solve Time)*

Matrix	ILUT $(10^{-1}, f)$	ILUTI $(10^{-1}, f)$	ILUT $(10^{-3}, f)$	ILUTI $(10^{-3}, f)$	ILUT $(0, k)$	ILUTI $(0, k)$	WSO
1	*F*	20(0.53)	*F*	6(0.58)	19(2.87)	1(2.13)	1(0.33)
2	12(2.22)	11(2.20)	$> 10min$	$> 10min$	$> 10min$	$> 10min$	23(1.8)
3	*F*	2(29.92)	*F*	2(29.94)	*F*	2(219.2)	7(10.32)
4	6(0.76)	7(0.31)	5(35.12)	3(2.07)	6(70.51)	2(2.64)	19(1.18)
5	18(9.5)	40(6.82)	8(12.94)	14(8.17)	$> 10min$	$> 10min$	25(5.54)
6	*Div.*	*Div.*	500(3.71)*	*Stag.*	*Div.*	1(1.09)	16(0.52)
7	9(0.09)	11(0.16)	3(0.19)	3(0.20)	2(13.19)	2(4.94)	37(1.09)
8	5(0.12)	5(0.2)	2(0.54)	2(0.32)	2(5.6)	1(4.52)	8(1.74)
9	*F*	*Div.*	1(2.1)	*Div.*	1(22.65)	*Div.*	1(0.7)
10	37(2.15)	37(2.4)	20(5.99)	19(6.18)	$> 10min$	$> 10min$	151(16.8)
11	*F*	*Div.*	*Div.*	*Div.*	*Div.*	*Div.*	500(20.04)*
12	7(0.38)	4(0.39)	*Div.*	*Div.*	*Div.*	5(12.68)	6(0.65)

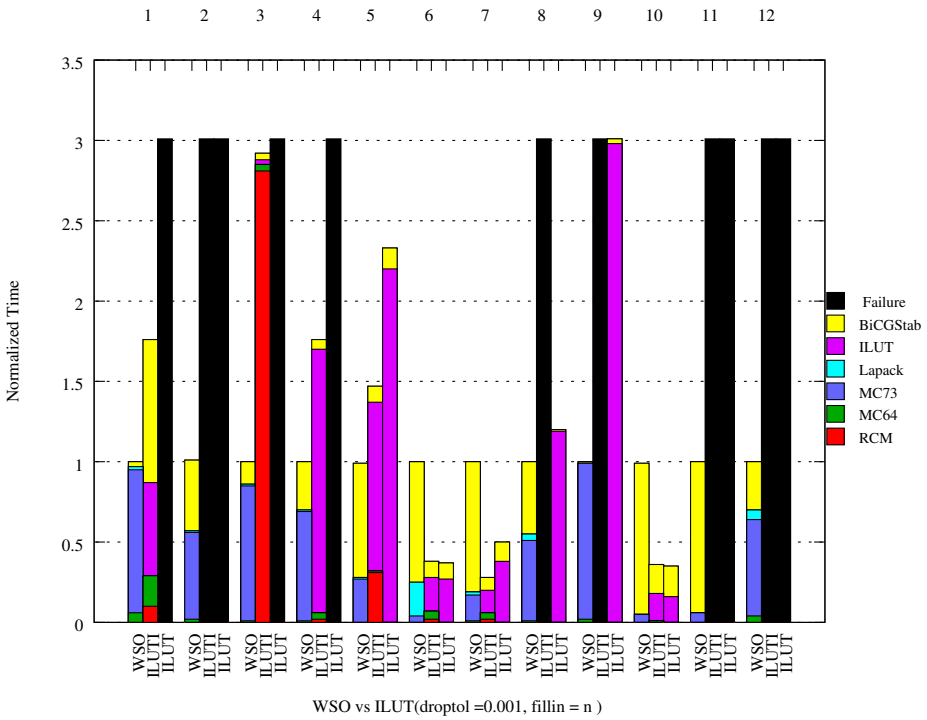


Figure 3.2: *Comparison of performance of WSO banded preconditioner with* $ILUT(f_{max}, 10^{-3})$

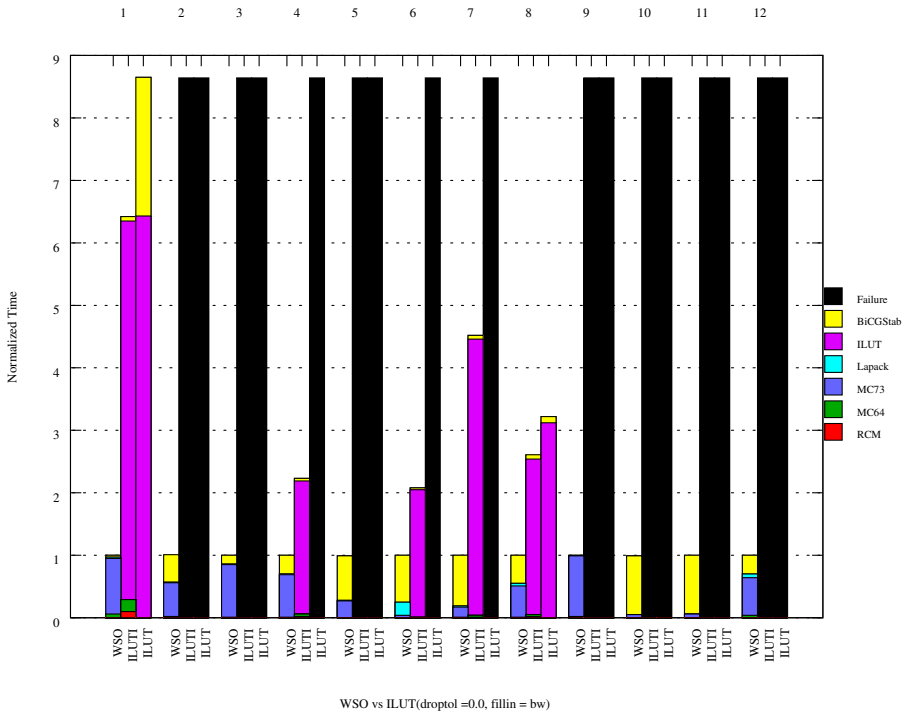


Figure 3.3: *Comparison of performance of WSO banded preconditioner with ILUT(k,* 0.0*)*

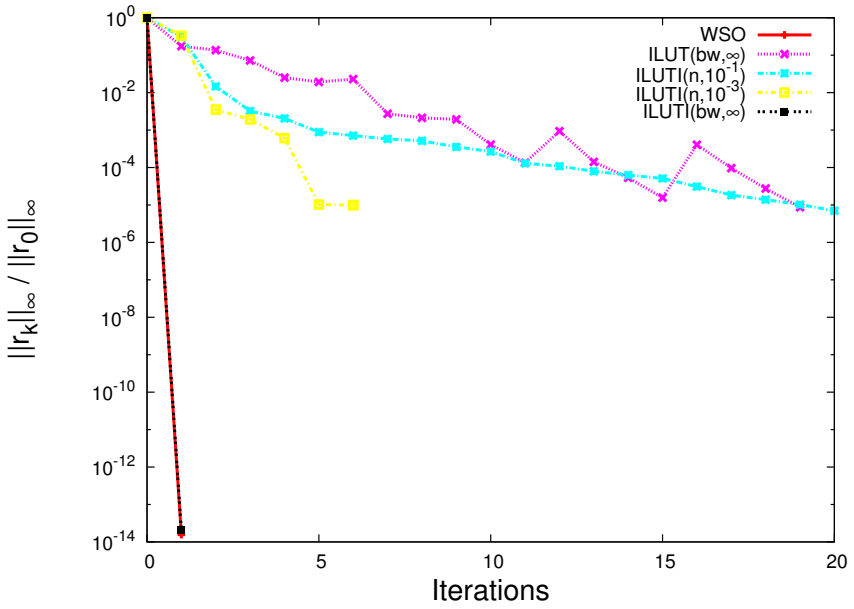


Figure 3.4: *Residual history of WSO banded preconditioner for problem 2D_54019_HIGHK*

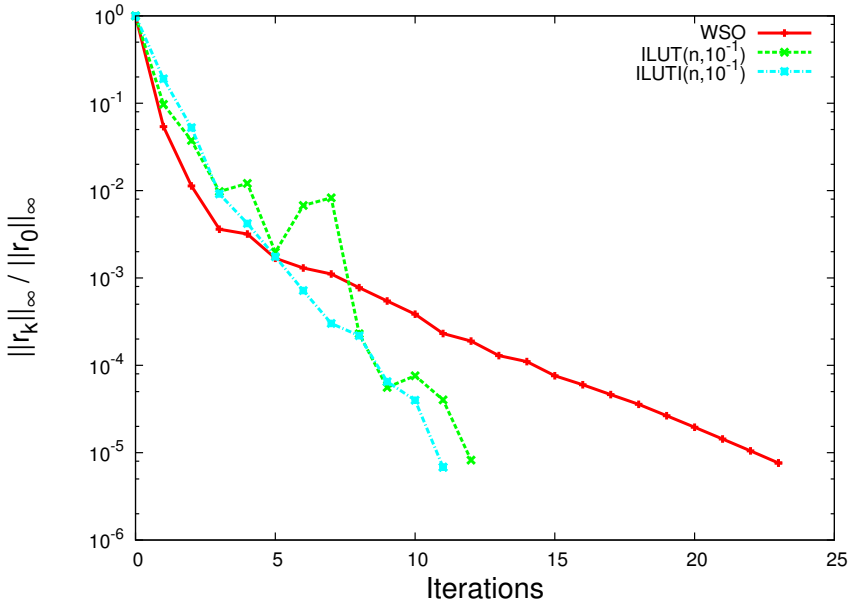


Figure 3.5: *Residual history of WSO banded preconditioner for problem Appu*

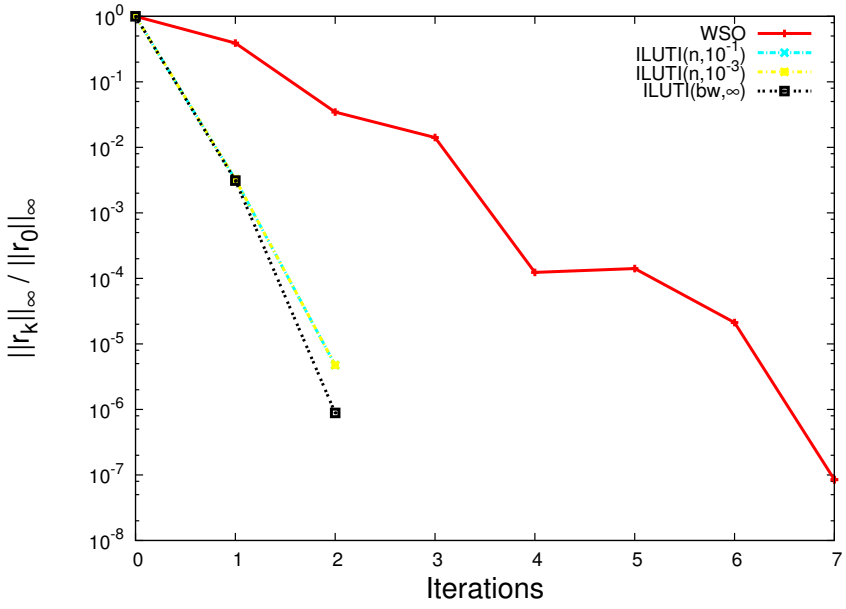


Figure 3.6: *Residual history of WSO banded preconditioner for problem ASIC_680k*

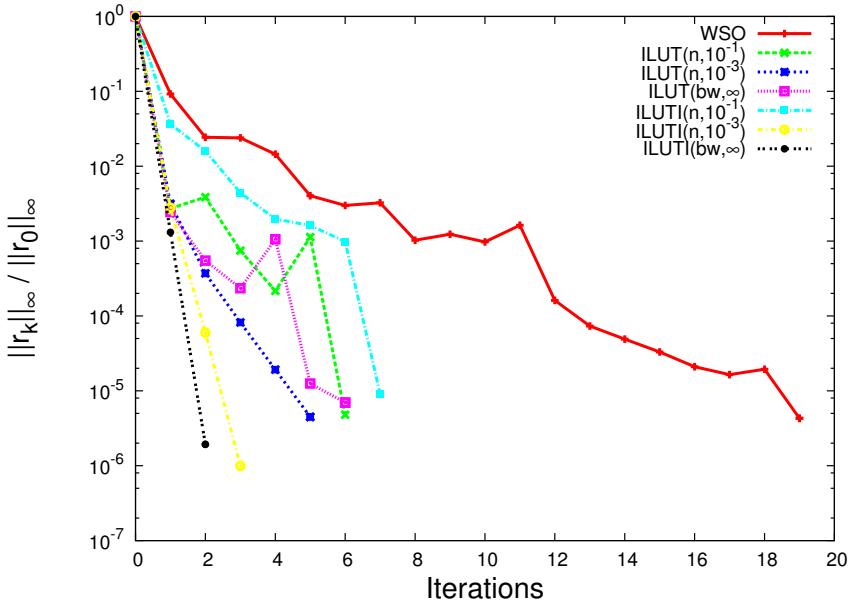


Figure 3.7: *Residual history of WSO banded preconditioner for problem BUNDLE1*

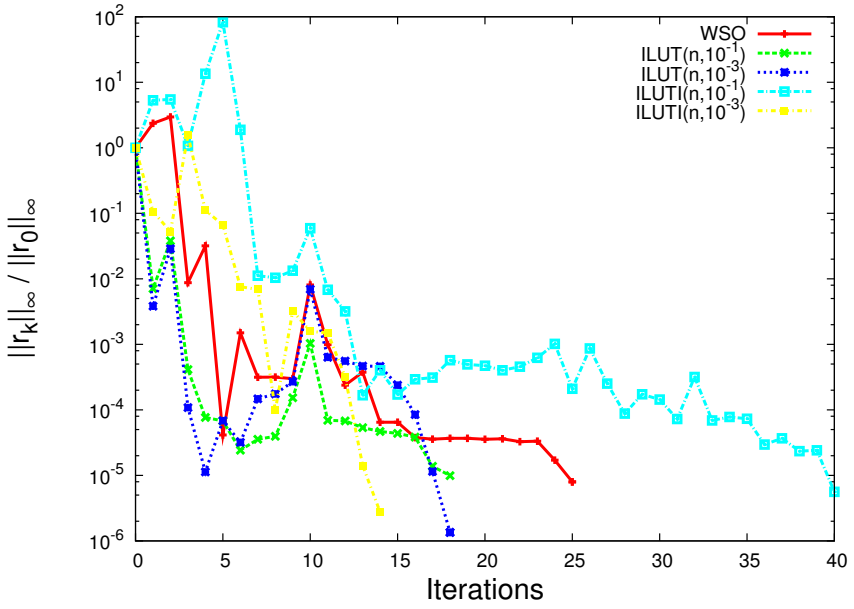


Figure 3.8: *Residual history of WSO banded preconditioner for problem DC1*

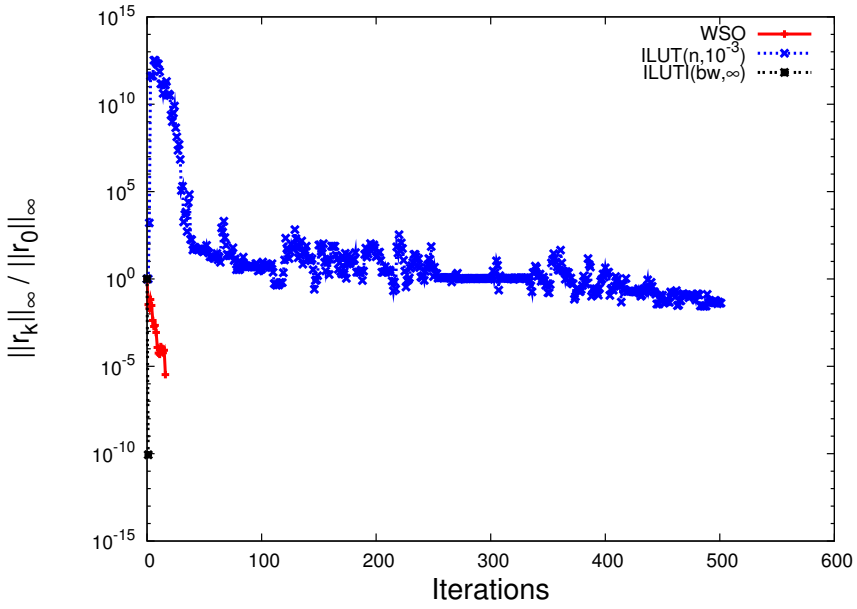


Figure 3.9: *Residual history of WSO banded preconditioner for problem DW8192*

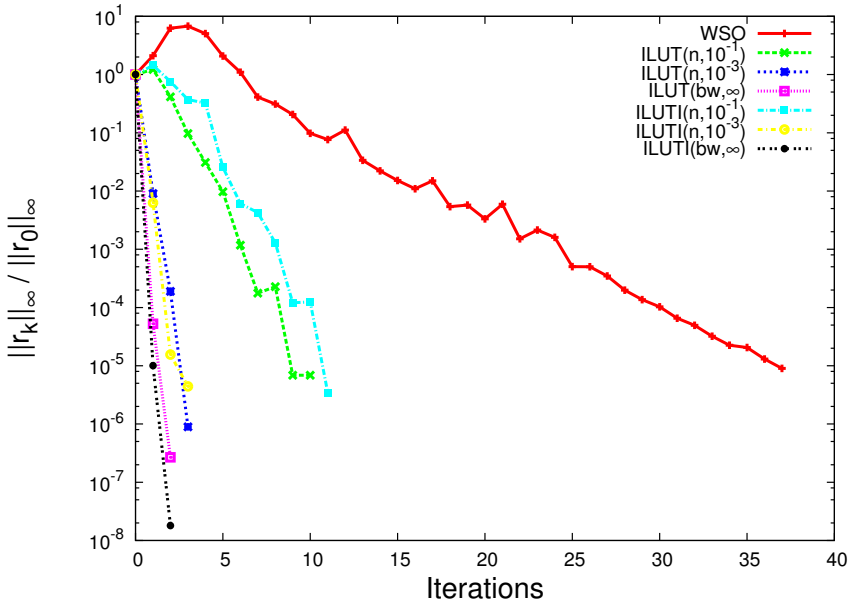


Figure 3.10: *Residual history of WSO banded preconditioner for problem FEM_3D_THERMAL*

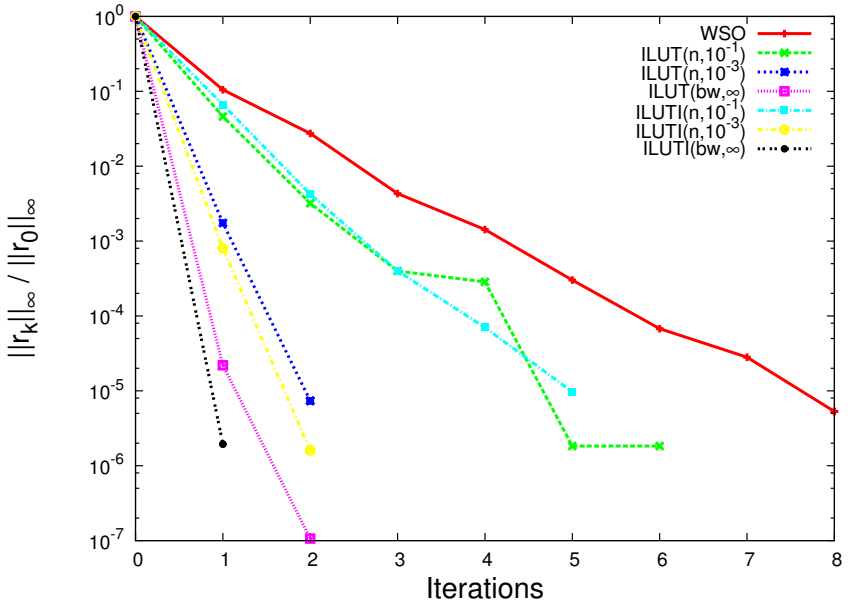


Figure 3.11: *Residual history of WSO banded preconditioner for problem FIDAP/EX5*

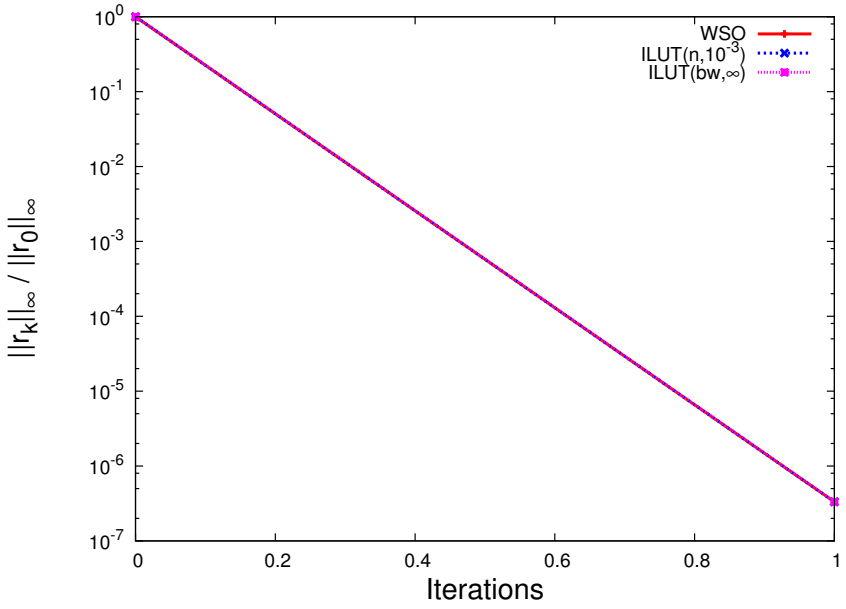


Figure 3.12: *Residual history of WSO banded preconditioner for problem FP*

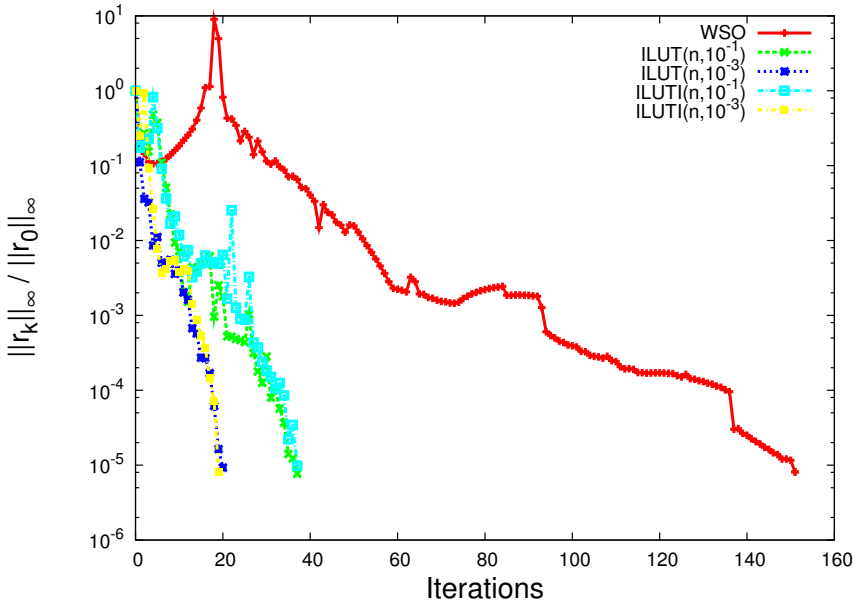


Figure 3.13: *Residual history of WSO banded preconditioner for problem H2O*

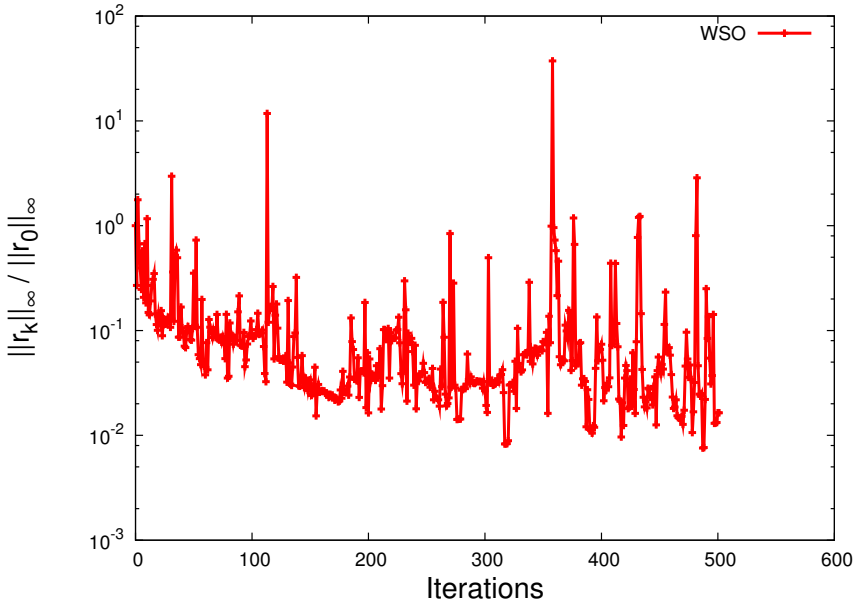


Figure 3.14: *Residual history of WSO banded preconditioner for problem MSC23052*

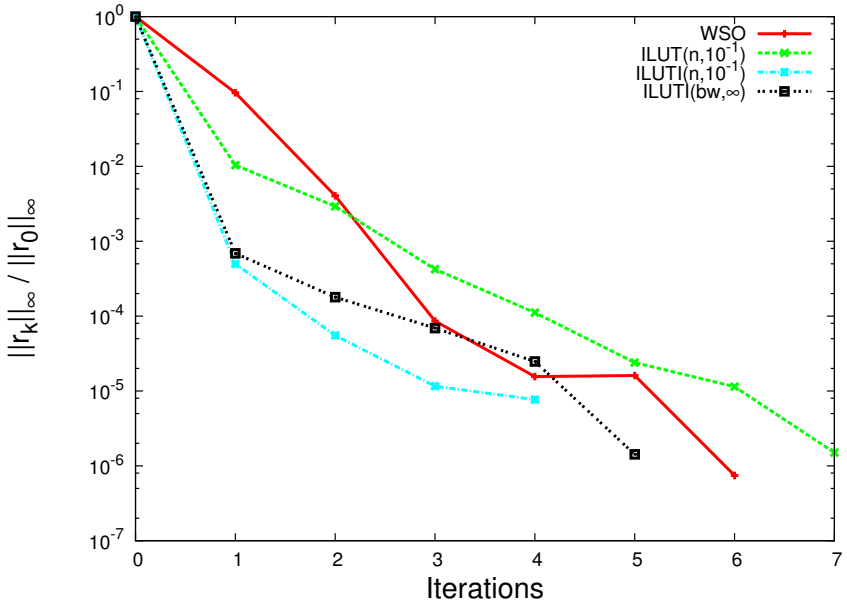


Figure 3.15: *Residual history of WSO banded preconditioner for problem RAEFSKY4*

Chapter 4

A Tearing-based Hybrid Parallel Banded Linear System Solver

4.1 Introduction

Numerical handling of partial differential equations (PDEs) plays a crucial role in modeling of physical processes. It involves discretization of these PDEs using, for example, Finite Difference or Finite Element methods and results in nonlinear systems of equations whose solution yields at each iteration a large sparse linear system. These systems can often be reordered using Reverse Cuthill-McKee [20, 53] or Spectral [29, 30, 72] reorderings into banded or low-rank perturbations of banded linear systems and solved using direct or preconditioned iterative methods in which the preconditioner is a banded matrix. In this Chapter we propose a novel hybrid parallel algorithm for solving banded linear systems and state conditions that guarantee its convergence.

The parallel solution of banded linear systems has been considered by many authors [19, 23, 24, 34, 37, 47, 55, 59, 60, 67, 81]. The overarching strategy consists of two main stages: (i) the coefficient matrix is reordered or modified so that it consists of several independent blocks, but which are interconnected by a single block. Certain algorithms produce this single block as a Schur complement, others produce different reduced systems; (ii) once this reduced system corresponding to this single block is solved, the original problem decomposes into several independent smaller problems facilitating almost perfect parallelism in retrieving the rest of the solution vector.

Our approach is different from the algorithms cited above. Its main idea was first proposed by Sameh et al., [54] in which the study was restricted to diagonally dominant symmetric positive definite linear systems. In this chapter we generalize it to non-symmetric linear systems without the requirement of

diagonal dominance. Furthermore, we show how to precondition the implicit balance system.

The rest of this chapter is organized as follows: First, we introduce the algorithm by showing how it "tears" the original system. Second, we analyze the conditions that guarantee the non-singularity of the balance system for any nonsingular original system. Further, we show that if the original system is Symmetric Positive Definite (SPD) then the much smaller balance system is SPD as well. Third, we discuss preconditioned iterative methods for solving the balance system, the Conjugate Gradient (CG) for the SPD case, and the Stabilized Bi-Conjugate Gradient (BiCGStab) for the non-symmetric case. We call our algorithm Domain Decomposition CG (DDCG) or Domain Decomposition BiCGStab (DDBiCGStab) for symmetric and non-symmetric linear systems, respectively. Finally, we present numerical experiments and pseudocode.

4.2 Partitioning

We are interested in solving linear systems

$$A\mathbf{x} = \mathbf{f} \tag{4.1}$$

where $A \in \mathbb{R}^{n\times n}$ and $\mathbf{x}, \mathbf{f} \in \mathbb{R}^n$. Let $A = [a_{ij}]$ have a lower and upper bandwidths of kd, in other words $|i-j| < kd \ll n$. We can rewrite our banded linear system (4.1) using a block tridiagonal matrix A, blocked vectors $\mathbf{x}$ and $\mathbf{f}$.

For clarity of presentation, we illustrate the partitioning and tearing scheme using three partitions ($p = 3$). For the same reason some of our theorems are proven only for three partitions. Also, we assume that all the partitions are of equal size, m, and that all the overlaps are of identical size $\tau = kd$. Generalization to the case of $p > 3$ partitions of different sizes is straightforward.

The banded matrix A and vectors $\mathbf{x}$ and $\mathbf{f}$ can be written as

$$A = \begin{pmatrix} A_{11} & A_{12} & & & & & \\ A_{21} & A_{22} & A_{23} & & & & \\ & A_{32} & A_{33} & A_{34} & & & \\ & & A_{43} & A_{44} & A_{45} & & \\ & & & A_{54} & A_{55} & A_{56} & \\ & & & & A_{65} & A_{66} & A_{67} \\ & & & & & A_{76} & A_{77} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \\ \mathbf{x}_6 \\ \mathbf{x}_7 \end{pmatrix}, \tag{4.2}$$

$$\text{and} \quad \mathbf{f} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_3 \\ \mathbf{f}_4 \\ \mathbf{f}_5 \\ \mathbf{f}_6 \\ \mathbf{f}_7 \end{pmatrix}$$

Here, A_{ij}, $\mathbf{x}_i$ and $\mathbf{f}_i$ for $i, j = 1, \ldots, 7$ are blocks of appropriate sizes. Let us define the partitions of A, delineated by lines in the illustration of A in (4.2), as

$$A_k = \begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} & \\ A_{21}^{(k)} & A_{22}^{(k)} & A_{23}^{(k)} \\ & A_{32}^{(k)} & A_{33}^{(k)} \end{pmatrix} \quad \text{for } k = 1, 2, 3 \tag{4.3}$$

If $\eta = 2(k-1)$, then the blocks $A_{\mu\nu}^{(k)} = A_{\eta+\mu\eta+\nu}$, for $\mu, \nu = 1, 2, 3$, except for the top left $\mu = \nu \neq 1$ block of the last and middle partition, and bottom right $\mu = \nu \neq 3$ block of the first and middle partition. For these blocks the following equality holds $A_{33}^{(k-1)} + A_{11}^{(k)} = A_{\eta+1,\eta+1}$. The exact choice of splitting $A_{33}^{(k-1)}$ and $A_{11}^{(k)}$ will be discussed below.

Thus, we can rewrite (4.1) as 3 independent linear systems, $k = 1, 2, 3$,

$$\begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} & \\ A_{21}^{(k)} & A_{22}^{(k)} & A_{23}^{(k)} \\ & A_{32}^{(k)} & A_{33}^{(k)} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1^{(k)} \\ \mathbf{x}_2^{(k)} \\ \mathbf{x}_3^{(k)} \end{pmatrix} = \begin{pmatrix} (1-\alpha_{k-1})\,\mathbf{f}_{\eta+1} - \mathbf{y}_{k-1} \\ \mathbf{f}_{\eta+2} \\ \alpha_k \mathbf{f}_{\eta+3} + \mathbf{y}_k \end{pmatrix} \tag{4.4}$$

where $\mathbf{y}_k$ and α_k are yet to be specified. Now, we need to choose the scaling parameters $0 \leq \alpha_1, \alpha_2 \leq 1$ and the adjustment vector $\mathbf{y}^T = \left(\mathbf{y}_1^T, \mathbf{y}_2^T\right)$ so that the solution of (4.4) coincides with the respective parts of the solution of (4.1). We will achieve this if

$$\mathbf{x}_3^{(1)} = \mathbf{x}_1^{(2)} \text{ and } \mathbf{x}_3^{(2)} = \mathbf{x}_1^{(3)} \tag{4.5}$$

Note that $\alpha_0 = 0$, $\alpha_3 = 1$, $\mathbf{y}_0 = \mathbf{y}_3 = \mathbf{0}$, and without loss of generality, we choose $\alpha_1 = \alpha_2 = 0.5$. Let

$$A_k^{-1} = \begin{pmatrix} B_{11}^{(k)} & B_{12}^{(k)} & B_{13}^{(k)} \\ B_{21}^{(k)} & B_{22}^{(k)} & B_{23}^{(k)} \\ B_{31}^{(k)} & B_{32}^{(k)} & B_{33}^{(k)} \end{pmatrix} \tag{4.6}$$

we can rewrite (4.4) as

$$\begin{pmatrix} \mathbf{x}_1^{(k)} \\ \mathbf{x}_2^{(k)} \\ \mathbf{x}_3^{(k)} \end{pmatrix} = \begin{pmatrix} B_{11}^{(k)} & B_{12}^{(k)} & B_{13}^{(k)} \\ B_{21}^{(k)} & B_{22}^{(k)} & B_{23}^{(k)} \\ B_{31}^{(k)} & B_{32}^{(k)} & B_{33}^{(k)} \end{pmatrix} \begin{pmatrix} (1-\alpha_{k-1})\mathbf{f}_{\eta+1} - \mathbf{y}_{k-1} \\ \mathbf{f}_{\eta+2} \\ \alpha_k \mathbf{f}_{\eta+3} + \mathbf{y}_k \end{pmatrix} \tag{4.7}$$

obtaining

$$\begin{cases} \mathbf{x}_1^{(k)} = B_{11}^{(k)}((1-\alpha_{k-1})\mathbf{f}_{\eta+1} - \mathbf{y}_{k-1}) + B_{12}^{(k)}\mathbf{f}_{\eta+2} + B_{13}^{(k)}(\alpha_k \mathbf{f}_{\eta+3} + \mathbf{y}_k) \\ \mathbf{x}_3^{(k)} = B_{31}^{(k)}((1-\alpha_{k-1})\mathbf{f}_{\eta+1} - \mathbf{y}_{k-1}) + B_{32}^{(k)}\mathbf{f}_{\eta+2} + B_{33}^{(k)}(\alpha_k \mathbf{f}_{\eta+3} + \mathbf{y}_k) \end{cases} \tag{4.8}$$

Then, using (4.5) and (4.8) we obtain

$$\left(B_{33}^{(\zeta)} + B_{11}^{(\zeta+1)}\right)\mathbf{y}_\zeta = \mathbf{g}_\zeta + B_{31}^{(\zeta)}\mathbf{y}_{\zeta-1} + B_{13}^{(\zeta+1)}\mathbf{y}_{\zeta+1} \tag{4.9}$$

for $\zeta = 1, 2$, where

$$\mathbf{g}_\zeta = \left((\alpha_{\zeta-1} - 1)\, B_{31}^{(\zeta)}, -B_{32}^{(\zeta)}, (1-\alpha_\zeta)\, B_{11}^{(\zeta+1)} - \alpha_\zeta B_{33}^{(\zeta)}, B_{12}^{(\zeta+1)}, \alpha_{\zeta+1} B_{13}^{(\zeta+1)} \right) \begin{pmatrix} \mathbf{f}_{\eta-1} \\ \mathbf{f}_\eta \\ \mathbf{f}_{\eta+1} \\ \mathbf{f}_{\eta+2} \\ \mathbf{f}_{\eta+3} \end{pmatrix} \tag{4.10}$$

Finally, letting $\mathbf{g}^T = \left(\mathbf{g}_1^T, \mathbf{g}_2^T\right)$, the adjustment vector $\mathbf{y}$ can be found by solving the balance system

$$M\mathbf{y} = \mathbf{g} \tag{4.11}$$

where

$$M = \begin{pmatrix} B_{33}^{(1)} + B_{11}^{(2)} & -B_{13}^{(2)} \\ -B_{31}^{(2)} & B_{33}^{(2)} + B_{11}^{(3)} \end{pmatrix} \tag{4.12}$$

Once $\mathbf{y}$ is found, we can solve the three independent linear systems (4.4) in parallel. Next, we focus our attention on solving (4.11). First, we note that the matrix M is not available explicitly. Thus using a direct method to solve the balance system (4.12) is not possible and hence we need to resort to iterative schemes. Our iterative methods of choice for M are CG and BiCGStab, respectively. However, to use them, we must be able to compute the initial residual $\mathbf{r}_0 = \mathbf{g} - M\mathbf{y}_0$ and compute matrix vector products of the form $\mathbf{q} = M\mathbf{p}$. Also, we need to determine what conditions A and the partitions A_k must satisfy for M to be nonsingular. This will be addressed in the next section.

As a final note, we mention that in general (for arbitrary p) the balance system is block tridiagonal of the form

$$\begin{pmatrix} B_{33}^{(1)} + B_{11}^{(2)} & -B_{13}^{(2)} & & & \\ -B_{31}^{(2)} & B_{33}^{(2)} + B_{11}^{(3)} & -B_{13}^{(3)} & & \\ & & \ddots & & \\ & & -B_{31}^{(p-2)} & B_{33}^{(p-2)} + B_{11}^{(p-1)} & -B_{13}^{(p-1)} \\ & & & -B_{31}^{(p-1)} & B_{33}^{(p-1)} + B_{11}^{(p)} \end{pmatrix} \tag{4.13}$$

4.3 The Balance System

4.3.1 The symmetric positive definite case

In this section, we assume that A is SPD, and investigate the conditions under which the balance system is also SPD.

Theorem 1 *If partitions A_k in* (4.3) *are SPD for $k = 1, \ldots, p$ then the coefficient matrix M, of the balance system in* (4.11)*, is SPD.*

Let $p = 3$. Notice that if A_k is SPD then A_k^{-1} is also SPD. Let

$$Q^T = \begin{pmatrix} I & 0 & 0 \\ 0 & 0 & -I \end{pmatrix}. \tag{4.14}$$

Pre-multiplying (4.6) by P^T and post-multiplying by P, we obtain

$$Q^T A_k^{-1} Q = \begin{pmatrix} B_{11}^{(k)} & -B_{13}^{(k)} \\ -B_{31}^{(k)} & B_{33}^{(k)} \end{pmatrix}, \tag{4.15}$$

which is also SPD. However, since the matrix M can be written as the sum

$$\begin{aligned} M &= M_1 + M_2 + M_3 && (4.16) \\ &= \begin{pmatrix} B_{33}^{(1)} & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} B_{11}^{(2)} & -B_{13}^{(2)} \\ -B_{31}^{(2)} & B_{33}^{(2)} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & B_{11}^{(3)} \end{pmatrix}, && (4.17) \end{aligned}$$

$\mathbf{z}^T M \mathbf{z} > 0$ for any nonzero $\mathbf{z}$.

Let A be SPD and DD, then we can find a splitting that results in SPD/DD A_k, which guarantees that M is SPD. This result is contained in the following theorem.

Theorem 2 *If A in* (4.1) *is SPD and DD then the partitions A_k in* (4.3) *can be chosen such that they inherit the same properties. Further, the coefficient matrix M, of the resulting balance system in* (4.11)*, is SPD.*

Since A is DD, we only need to obtain a splitting that ensures the diagonal dominance of the overlapping parts. Let $\mathbf{e} = [1, \ldots, 1]^T$, $\mathbf{e}_i$ be the i-th column of the identity, $|.|$ denote the absolute value, diag(.) and offdiag(.) denote the diagonal and off diagonal elements, respectively. Now let the elements of the diagonal matrices $H_\zeta^{(1)} = [h_{ii}^{(\zeta,1)}]$ and $H_{\zeta+1}^{(2)} = [h_{ii}^{(\zeta+1,2)}]$, of appropriate sizes, be given by

$$\begin{aligned} h_{ii}^{(\zeta,1)} &= \mathbf{e}_i^T |A_{32}^{(\zeta)}| \mathbf{e} \quad + \frac{1}{2} \mathbf{e}_i^T |\text{offdiag}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)})| \mathbf{e} && (4.18) \\ h_{ii}^{(\zeta+1,2)} &= \mathbf{e}_i^T |A_{12}^{(\zeta+1)}| \mathbf{e} + \frac{1}{2} \mathbf{e}_i^T |\text{offdiag}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)})| \mathbf{e} && (4.19) \\ & && (4.20) \end{aligned}$$

respectively. Notice that $h_{ii}^{(\zeta,1)}$ and $h_{ii}^{(\zeta+1,2)}$ are sums of absolute values of all off diagonal elements, with elements in the overlap being halved, in the i-th row to the left and right of the diagonal, respectively. Moreover, let the difference between the positive diagonal elements and the sums of absolute values of all off diagonal elements in the same row be given by:

$$D_\zeta = \text{diag}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)}) - H_\zeta^{(1)} - H_{\zeta+1}^{(2)}. \tag{4.21}$$

Now, if

$$\begin{aligned} A_{33}^{(\zeta)} &= H_{\zeta}^{(1)} + \frac{1}{2}D_{\zeta} + \frac{1}{2}\text{offdiag}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)}), \text{ and} \\ A_{11}^{(\zeta+1)} &= H_{\zeta+1}^{(2)} + \frac{1}{2}D_{\zeta} + \frac{1}{2}\text{offdiag}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)}), \end{aligned} \tag{4.22}$$

it is easy to verify that $A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)} = A_{2\zeta+1,2\zeta+1}$ and each A_k, for $k = 1, \ldots, p$, is SPD/DD. Consequently, if (4.2) is SPD/DD so are the partitions A_k and by Theorem 1, the balance system is guaranteed to be SPD.

4.3.2 The non-symmetric case

Next, if A is just a nonsingular non-symmetric matrix, we explore conditions under which (4.12) becomes nonsingular.

Theorem 3 *Let the matrix A in* (4.1) *be any nonsingular matrix with partitions A_k, $k = 1, \ldots, p$, in* (4.3) *that are also nonsingular. Then the coefficient matrix M, of the balance system in* (4.11)*, is nonsingular.*

Let $p = 3$. Notice that we can write

$$A = \begin{pmatrix} A_1 & & \\ & 0_{m-2\tau} & \\ & & A_3 \end{pmatrix} + \begin{pmatrix} 0_{m-\tau} & & \\ & A_2 & \\ & & 0_{m-\tau} \end{pmatrix} \tag{4.23}$$

Next, let the nonsingular matrix C be given by,

$$\begin{aligned} C &= \begin{pmatrix} A_1^{-1} & & \\ & I_{m-2\tau} & \\ & & A_3^{-1} \end{pmatrix} A \begin{pmatrix} I_{m-\tau} & & \\ & A_2^{-1} & \\ & & I_{m-\tau} \end{pmatrix} \\ &= \begin{pmatrix} I_m & & \\ & 0_{m-2\tau} & \\ & & I_m \end{pmatrix} \begin{pmatrix} I_{m-\tau} & & \\ & A_2^{-1} & \\ & & I_{m-\tau} \end{pmatrix} \\ &\quad + \begin{pmatrix} A_1^{-1} & & \\ & I_{m-2\tau} & \\ & & A_3^{-1} \end{pmatrix} \begin{pmatrix} 0_{m-\tau} & & \\ & I_m & \\ & & 0_{m-\tau} \end{pmatrix} \end{aligned} \tag{4.24}$$

where I_m and 0_m are the identity and zero matrices of order m, respectively. Using (4.23), (4.24), (4.6), we obtain

$$
\begin{aligned}
C &= \begin{pmatrix}
I_\tau & & & & & & & \\
& I_{m-2\tau} & & & & & & \\
& & B_{11}^{(2)} & B_{12}^{(2)} & B_{13}^{(2)} & & & \\
& & 0 & 0_{m-2\tau} & 0 & & & \\
& & B_{31}^{(2)} & B_{32}^{(2)} & B_{33}^{(2)} & & & \\
& & & & & I_{m-2\tau} & & \\
& & & & & & I_\tau &
\end{pmatrix} \\
&\quad + \begin{pmatrix}
0_\tau & 0 & B_{13}^{(1)} & & & & & \\
0 & 0_{m-2\tau} & B_{23}^{(1)} & & & & & \\
0 & 0 & B_{33}^{(1)} & & & & & \\
& & & I_{m-2\tau} & & & & \\
& & & & B_{11}^{(3)} & 0 & 0 & \\
& & & & B_{21}^{(3)} & 0_{m-2\tau} & 0 & \\
& & & & B_{31}^{(3)} & 0 & 0_\tau &
\end{pmatrix} \\
&= \begin{pmatrix}
I_\tau & 0 & B_{13}^{(1)} & & & & & \\
0 & I_{m-2\tau} & B_{23}^{(1)} & & & & & \\
0 & 0 & B_{33}^{(1)} + B_{11}^{(2)} & B_{12}^{(2)} & B_{13}^{(2)} & & & \\
& & 0 & I_{m-2\tau} & 0 & & & \\
& & B_{31}^{(2)} & B_{32}^{(2)} & B_{33}^{(2)} + B_{11}^{(3)} & 0 & 0 & \\
& & & & B_{21}^{(3)} & I_{m-2\tau} & 0 & \\
& & & & B_{31}^{(3)} & 0 & I_\tau &
\end{pmatrix}
\end{aligned}
$$

where the 0 matrices without subscripts are considered to be of appropriate sizes. Using the permutation matrix

$$
P^T = \begin{pmatrix}
I_\tau & & & & & & \\
& I_{m-2\tau} & & & & & \\
& & I_{m-2\tau} & & & & \\
& & & & I_{m-2\tau} & & \\
& & & & & I_\tau & \\
& & I_\tau & & & & \\
& & & -I_\tau & & &
\end{pmatrix} \tag{4.25}
$$

we can write

$$P^T C P = \left(\begin{array}{ccccc|cc} I_\tau & & & & & B_{13}^{(1)} & \\ & I_{m-2\tau} & & & & B_{23}^{(1)} & \\ & & I_{m-2\tau} & & & & \\ & & & I_{m-2\tau} & & & -B_{21}^{(3)} \\ & & & & I_\tau & & -B_{31}^{(3)} \\ \hline & B_{12}^{(2)} & & & & B_{33}^{(1)}+B_{11}^{(2)} & -B_{13}^{(2)} \\ & -B_{32}^{(2)} & & & & -B_{31}^{(2)} & B_{33}^{(2)}+B_{11}^{(3)} \end{array}\right) \tag{4.26}$$

or rewriting (4.26) as a block 2×2 matrix, with blocks delineated by lines above, we obtain

$$P^T C P = \left(\begin{array}{c|c} I_{3m-4\tau} & Z_1 \\ \hline Z_2^T & M \end{array}\right). \tag{4.27}$$

Consider the eigenvalue problem

$$\left(\begin{array}{c|c} I_{3m-4\tau} & Z_1 \\ \hline Z_2^T & M \end{array}\right)\left(\begin{array}{c} \mathbf{u}_1 \\ \mathbf{u}_2 \end{array}\right) = \lambda \left(\begin{array}{c} \mathbf{u}_1 \\ \mathbf{u}_2 \end{array}\right). \tag{4.28}$$

Pre-multiplying the first block row of (4.28) by Z_2^T and noticing that $Z_2^T Z_1 = 0$ we obtain

$$(1-\lambda) Z_2^T \mathbf{u}_1 = 0 \tag{4.29}$$

Hence, either $\lambda = 1$ or $Z_2^T \mathbf{u}_1 = 0$. If $\lambda \neq 1$, then the second block row of (4.28) yields

$$M \mathbf{u}_2 = \lambda \mathbf{u}_2 \tag{4.30}$$

Thus, the eigenvalues λ of C are either 1 or are identical to those of the balance system, and we can write:

$$\lambda(C) = \lambda(P^T C P) \in \{1, \lambda(M)\}. \tag{4.31}$$

Hence, since C is nonsingular, the balance system is also nonsingular.

Notice that (4.24) and (4.31) suggest a powerful preconditioning technique for the banded linear system (4.1).

Next, we explore conditions that the nonsingular matrix A must satisfy, so that we are guaranteed that there exists a splitting that results in nonsingular partitions A_k. We also provide a formula for computing such a splitting.

Theorem 4 *If matrix A in (4.1) is nonsingular, its symmetric part $H = \frac{1}{2}(A + A^T)$ is symmetric positive semidefinite (SPSD) and $\mathcal{N}(H) \cap \mathcal{N}(B) = \oslash$ for $B = B_1 = B_2^T$ where*

$$B_1 = \begin{pmatrix} A_{11}^{(2)} & & & & & \\ A_{21}^{(2)} & & & & & \\ & A_{11}^{(3)} & & & & \\ & A_{21}^{(3)} & & & & \\ & & \ddots & & & \\ & & & & A_{11}^{(p)} & \\ & & & & A_{21}^{(p)} & \end{pmatrix}, B_2^T = \begin{pmatrix} A_{11}^{(2)^T} & & & & & \\ A_{12}^{(2)^T} & & & & & \\ & A_{11}^{(3)^T} & & & & \\ & A_{12}^{(3)^T} & & & & \\ & & \ddots & & & \\ & & & & A_{11}^{(p)^T} & \\ & & & & A_{12}^{(p)^T} & \end{pmatrix}, \tag{4.32}$$

then there exists a splitting such that the partitions A_k in (4.3) for $k = 1, \ldots, p$ are nonsingular.

Let $p = 3$. Let $\widetilde{A}$ be the block diagonal matrix in which the blocks are the partitions A_k,

$$\widetilde{A} = \begin{pmatrix} A_{11}^{(1)} & A_{12}^{(1)} & & & & & & & & \\ A_{21}^{(1)} & A_{22}^{(1)} & A_{23}^{(1)} & & & & & & & \\ & A_{32}^{(1)} & A_{33}^{(1)} & & & & & & & \\ & & & A_{11}^{(2)} & A_{12}^{(2)} & & & & & \\ & & & A_{21}^{(2)} & A_{22}^{(2)} & A_{23}^{(2)} & & & & \\ & & & & A_{32}^{(2)} & A_{33}^{(2)} & & & & \\ & & & & & & A_{11}^{(3)} & A_{12}^{(3)} & & \\ & & & & & & A_{21}^{(3)} & A_{22}^{(3)} & A_{23}^{(3)} & \\ & & & & & & & A_{32}^{(3)} & A_{33}^{(3)} & \end{pmatrix} \tag{4.33}$$

and the nonsingular permutation matrix J^T be defined as

$$J^T = \begin{pmatrix} I_\tau & & & & & & & & & \\ & I_{m-2\tau} & & & & & & & & \\ & & I_\tau & I_\tau & & & & & & \\ & & & & I_{m-2\tau} & & & & & \\ & & & & & I_\tau & I_\tau & & & \\ & & & & & & & I_{m-2\tau} & & \\ & & & & & & & & I_\tau & \\ & & 0 & I_\tau & & & & & & \\ & & & & & 0 & I_\tau & & & \end{pmatrix} \tag{4.34}$$

Then, since $\operatorname{rank}(\widetilde{A}) = \operatorname{rank}(J^T \widetilde{A} J)$, where

$$J^T \widetilde{A} J = \left(\begin{array}{ccccccccc|cc}
A_{11}^{(1)} & A_{12}^{(1)} & & & & & & & & & \\
A_{21}^{(1)} & A_{22}^{(1)} & A_{23}^{(1)} & & & & & & & & \\
 & A_{32}^{(1)} & A_{33}^{(1)} + A_{11}^{(2)} & A_{12}^{(2)} & & & & & & A_{11}^{(2)} & \\
 & & A_{21}^{(2)} & A_{22}^{(2)} & A_{23}^{(2)} & & & & & A_{21}^{(2)} & \\
 & & & A_{32}^{(2)} & A_{33}^{(2)} + A_{11}^{(3)} & A_{12}^{(3)} & & & & & A_{11}^{(3)} \\
 & & & & A_{21}^{(3)} & A_{22}^{(3)} & A_{23}^{(3)} & & & & A_{21}^{(3)} \\
 & & & & & A_{32}^{(3)} & A_{33}^{(3)} & & & & \\
\hline
 & & A_{11}^{(2)} & A_{12}^{(2)} & & & & & & A_{11}^{(2)} & \\
 & & & & A_{11}^{(3)} & A_{12}^{(3)} & & & & & A_{11}^{(3)}
\end{array}\right). \tag{4.35}$$

This can be written as a block 2×2 matrix,

$$J^T \widetilde{A} J = \left(\begin{array}{c|c} A & \hat{B}_1 \\ \hline \hat{B}_2^T & K \end{array}\right), \tag{4.36}$$

where $\hat{B}_1$ and $\hat{B}_2$ contain B_1 and B_2, respectively, with additional zeros at the top and bottom. Notice that using splitting $A_{11}^{(k)} = \frac{1}{2}(A_{\eta+1\eta+1} + A_{\eta+1\eta+1}^T) + \beta I$ and $A_{33}^{(k-1)} = \frac{1}{2}(A_{\eta+1\eta+1} - A_{\eta+1\eta+1}^T) - \beta I$ we can choose β so as to ensure that $\hat{B}_1$ and $\hat{B}_2$ are of full rank and K is SPD. Thus, using Theorem 3.4 on p. 17 of [8], we conclude that $\widetilde{A}$ has full rank and consequently the partitions A_k are nonsingular.

Notice that the condition $B = B_1 = B_2^T$ in the theorem above is not as restrictive as it looks. Recalling that the original matrix is banded, it is easy to see that matrices $A_{12}^{(k)}$ and $A_{21}^{(k)}$ are almost completely zero except for small parts in the respective corners of size no larger than the overlap. This condition can be viewed as a requirement of symmetry surrounding the overlaps.

Let us now focus on two special cases. First, if the matrix A is SPD the conditions of the above theorem are immediately satisfied and we obtain the following Corollary.

Corollary 1 *If the matrix A in* (4.1) *is SPD then there exists a splitting (as described in Theorem 4) such that the partitions A_k in* (4.3) *for $k = 1, \ldots, p$ are nonsingular and consequently the coefficient matrix M, of the balance system in* (4.11)*, is nonsingular.*

Second, if the symmetry requirement is dropped in Theorem 2, then combining the results of Theorems 2 and 3 we obtain the following.

Corollary 2 *If A in* (4.1) *is DD (hence nonsingular) then the partitions A_k in* (4.3) *can be chosen such that they are also DD (hence nonsingular) for $k = 1, \ldots, p$ and consequently the coefficient matrix M, of the balance system in* (4.11)*, is nonsingular.*

4.4 The Hybrid Solver of the Balance System

Let us show how we can compute the initial residual $\mathbf{r}_0$ needed to start either CG or BiCGStab for solving the balance system. Notice that we can rewrite (4.7) as

$$\begin{pmatrix} \mathbf{x}_1^{(k)} \\ \mathbf{x}_2^{(k)} \\ \mathbf{x}_3^{(k)} \end{pmatrix} = \begin{pmatrix} \mathbf{h}_1^{(k)} \\ \mathbf{h}_2^{(k)} \\ \mathbf{h}_3^{(k)} \end{pmatrix} + \begin{pmatrix} \bar{\mathbf{y}}_1^{(k)} \\ \bar{\mathbf{y}}_2^{(k)} \\ \bar{\mathbf{y}}_3^{(k)} \end{pmatrix}, \tag{4.37}$$

where

$$\begin{pmatrix} \mathbf{h}_1^{(k)} \\ \mathbf{h}_2^{(k)} \\ \mathbf{h}_3^{(k)} \end{pmatrix} = A_k^{-1} \begin{pmatrix} (1-\alpha_{k-1})\mathbf{f}_{\eta+1} \\ \mathbf{f}_{\eta+2} \\ \alpha_k \mathbf{f}_{\eta+3} \end{pmatrix}, \quad \begin{pmatrix} \bar{\mathbf{y}}_1^{(k)} \\ \bar{\mathbf{y}}_2^{(k)} \\ \bar{\mathbf{y}}_3^{(k)} \end{pmatrix} = A_k^{-1} \begin{pmatrix} -\mathbf{y}_{k-1} \\ 0 \\ \mathbf{y}_k \end{pmatrix} \tag{4.38}$$

and the residual can be written as

$$\mathbf{r} = \mathbf{g} - M\mathbf{y} = \begin{pmatrix} \mathbf{x}_1^{(2)} - \mathbf{x}_3^{(1)} \\ \mathbf{x}_1^{(3)} - \mathbf{x}_3^{(2)} \\ \vdots \\ \mathbf{x}_1^{(p)} - \mathbf{x}_3^{(p-1)} \end{pmatrix} = \begin{pmatrix} \mathbf{h}_1^{(2)} - \mathbf{h}_3^{(1)} \\ \mathbf{h}_1^{(3)} - \mathbf{h}_3^{(2)} \\ \vdots \\ \mathbf{h}_1^{(p)} - \mathbf{h}_3^{(p-1)} \end{pmatrix} + \begin{pmatrix} \bar{\mathbf{y}}_1^{(2)} - \bar{\mathbf{y}}_3^{(1)} \\ \bar{\mathbf{y}}_1^{(3)} - \bar{\mathbf{y}}_3^{(2)} \\ \vdots \\ \bar{\mathbf{y}}_1^{(p)} - \bar{\mathbf{y}}_3^{(p-1)} \end{pmatrix} \tag{4.39}$$

Let the initial guess be given by $\mathbf{y}_0 = \mathbf{0}$, then we have

$$\mathbf{r}_0 = \mathbf{g} = \begin{pmatrix} \mathbf{h}_1^{(2)} - \mathbf{h}_3^{(1)} \\ \mathbf{h}_1^{(3)} - \mathbf{h}_3^{(2)} \\ \vdots \\ \mathbf{h}_1^{(p)} - \mathbf{h}_3^{(p-1)} \end{pmatrix}. \tag{4.40}$$

Thus, to compute the initial residual we must solve the p independent linear systems and subtract the bottom part of the solution vector of partition ζ, $\mathbf{h}_3^{(\zeta)}$, from the top part of the solution vector of partition $\zeta+1$, $\mathbf{h}_1^{(\zeta+1)}$, for $\zeta = 1, \ldots, p-1$.

Finally, to compute matrix-vector products, $\mathbf{q} = M\mathbf{p}$, using (4.39) and (4.40) we obtain

$$M\mathbf{y} = \mathbf{g} - \mathbf{r} = \mathbf{r}_0 - \mathbf{r} = \begin{pmatrix} \bar{\mathbf{y}}_3^{(1)} - \bar{\mathbf{y}}_1^{(2)} \\ \bar{\mathbf{y}}_3^{(2)} - \bar{\mathbf{y}}_1^{(3)} \\ \vdots \\ \bar{\mathbf{y}}_3^{(p-1)} - \bar{\mathbf{y}}_1^{(p)} \end{pmatrix} \tag{4.41}$$

Hence, we can compute matrix-vector products $M\mathbf{p}$, for any vector $\mathbf{p}$, in a similar fashion as the initial residual using (4.41) and (4.38).

The modified iterative methods (CG and BiCGStab) used to solve (4.11) are the standard iterative methods with initial residual and matrix-vector products computed using (4.40) and (4.41), respectively.

4.5 Preconditioning the balance system

We precondition the balance system using a block diagonal matrix of the form,

$$\widetilde{M} = \begin{pmatrix} \widetilde{B}_{33}^{(1)} + \widetilde{B}_{11}^{(2)} & & \\ & \ddots & \\ & & \widetilde{B}_{33}^{(p-1)} + \widetilde{B}_{11}^{(p)} \end{pmatrix} \tag{4.42}$$

where $B_{33}^{(\zeta)} \approx \widetilde{B}_{33}^{(\zeta)} = {A_{33}^{(\zeta)}}^{-1}$ and $B_{11}^{(\zeta+1)} \approx \widetilde{B}_{11}^{(\zeta+1)} = {A_{11}^{(\zeta+1)}}^{-1}$. Here, we are taking advantage of the fact that the elements of the inverse of a banded matrix decay as we move away from the diagonal, e.g. [22]. Also such decay becomes more pronounced as the banded matrix is more diagonally dominant. Using Sherman-Morrison-Woodbury formula, [36], we can write

$$\begin{aligned} (\widetilde{B}_{33}^{(\zeta)} + \widetilde{B}_{11}^{(\zeta+1)})^{-1} &= ({A_{33}^{(\zeta)}}^{-1} + {A_{11}^{(\zeta+1)}}^{-1})^{-1} = A_{33}^{(\zeta)}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)})^{-1}A_{11}^{(\zeta+1)} = \\ &= A_{33}^{(\zeta)} - A_{33}^{(\zeta)}(A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)})^{-1}A_{33}^{(\zeta)} \end{aligned} \tag{4.43}$$

where we prefer the last equality as it avoids extra interprocessor communication, assuming that the original overlapping block $A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)}$ is stored separately on processor ζ. Consequently, to precondition (4.11) we only need to multiply vectors with $A_{33}^{(\zeta)}$, and solve small linear systems with coefficient matrices $A_{33}^{(\zeta)} + A_{11}^{(\zeta+1)}$.

4.6 Numerical Experiments

We compare the performance of our hybrid solver with both preconditioned iterative solvers as well as direct solvers. Consequently, in this section, we compare DDCG and DDBiCGStab with preconditioned CG and BiCGStab, as well as LAPACK [1], and ScaLapack [12]. These solvers have been tested on a variety of linear systems. Here we present only few examples whose results are typical of a much larger collection. The six test problems are banded systems extracted from two large sparse matrices, selected from The University of Florida Sparse Matrix Collection [21] (see Table 4.1). First, these two matrices E_i for $i = 1, 2$ are reordered using the Reverse Cuthill-McKee scheme and three central bands of bandwidths 129, 257, and 513 are extracted from them. The diagonal elements of these banded systems are then perturbed so that the three symmetric matrices S_j are SPD/DD with a degree of DD $\backsim 1.008$, and the other three non-symmetric matrix N_j are made nonsingular. Finally, all six matrices are equilibrated to produce condition numbers of $\backsim 7.6\mathrm{E}+3$ for the matrices S_j, and $\backsim 4.1\mathrm{E}+6$ for the matrices N_j.

Notice that diagonal dominant is not a requirement for the convergence of DDCG; we only need the partitions to be SPD (see Theorem 1), nevertheless we ensure diagonal dominant to give an extra advantage to the block-Jacobi (BJ) preconditioned CG. Also, even though we did not prove the convergence of our

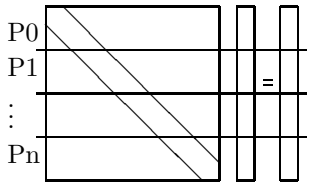


Figure 4.1: *Distribution of a banded linear system across various processors.*

hybrid solver assuming only non-singularity of the linear system, see Theorems 3 and 4, our hybrid solver proved to be successful in handling such a case.

Table 4.1: *Description of test matrices*

Matrix	Size	Nonzeros	Symm.	Application
E_1: AMD/G3_circuit	1,585,478	7,660,826	yes	Circuit Simulation
E_2: Sandia/ASIC_680k	682,862	4,001,317	no	Circuit Simulation

Our experiments are performed on an SGI Altix (with Intel Itanium 2 processors) at the National Center for Supercomputing Applications (NCSA) of the University of Illinois at Urbana-Champaign. For every run, we state the time <seconds>, <# of iterations> of the iterative scheme, and the 2-norm of the residual $< ||\mathbf{r}||_2 = ||\mathbf{f} - A\mathbf{x}||_2 >$. In all experiments, the exact solution is $\mathbf{x}^* = \mathbf{e}$, with the right-hand-side chosen as the multiplication of either S_j or N_j with $\mathbf{e}$. The six experiments are performed on 1, 2, 4, 8, 16 and 32 processors. The stopping criteria (s.c.) for the classical CG and BiCGstab are chosen as the relative residual $||\mathbf{r}||_2/||\mathbf{r}_0||_2 \leq 10^{-4}, 10^{-6}, 10^{-10}$, while for our hybrid solver the stopping criterion is chosen as $||\mathbf{r}||_2/||\mathbf{r}_0||_2 \leq 10^{-4}$. We also terminate all solvers if the number of iterations exceeds the size of the system to be solved.

For parallel implementation, the banded matrices and the corresponding right-hand-sides are distributed by blocks of rows across the processors (see Fig. 4.1). BJ preconditioning is chosen for CG and BiCGstab as it allows perfect parallelism in the preconditioning stage. It is worth mentioning that BJ preconditioning of CG and BiCGstab requires solving independent linear systems each of order equals to the number of rows on the current processor, while preconditioning our hybrid solver requires solving independent linear systems of the much smaller order which is equal to the bandwidth.

It can be seen from the experiments that preconditioning the balance system in DDCG and DDBiCGStab schemes significantly reduces the number of iterations. As mentioned earlier, this a great benefit for very little cost. Figures 4.2 and 4.3 show the 2-norms of the final residuals achieved by all four solvers. the lowest 2-norm of the residuals was achieved by our hybrid solver and ScaLa-

pack even though our stopping criterion for solving the balance system is only $||\mathbf{r}||_2/||\mathbf{r}_0||_2 \leq 10^{-4}$. Requiring such low residuals from the classical CG and BiCGstab would have consumed much more time than illustrated in 4.4 and 4.5. These two figures illustrate the excellent performance of our parallel hybrid solver which is enhanced as the bandwidth increases.

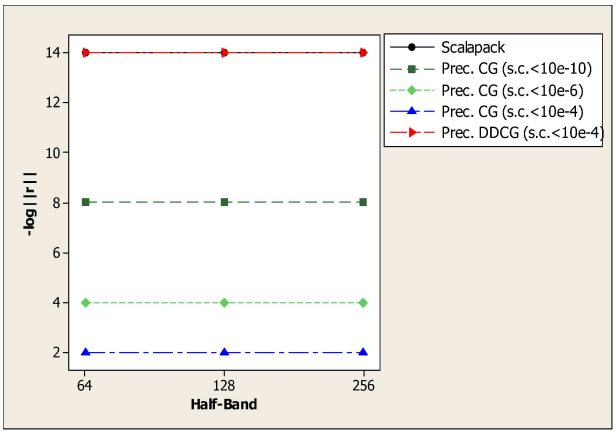


Figure 4.2: *Achieved residual for different methods and stopping criteria on G3_circuit*

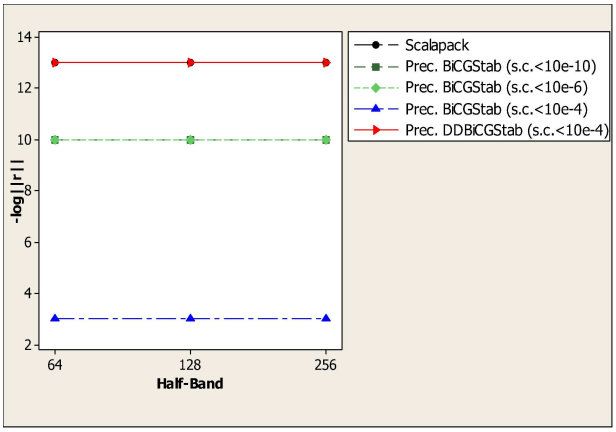


Figure 4.3: *Achieved residual for different methods and stopping criteria on ASIC_680k*

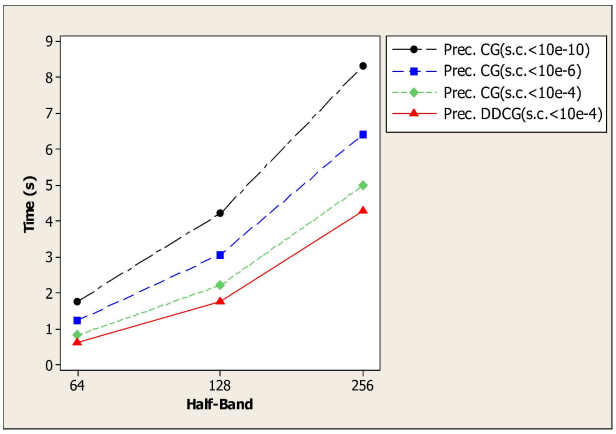


Figure 4.4: *Performance on 8 processors with different stopping criteria on G3_circuit*

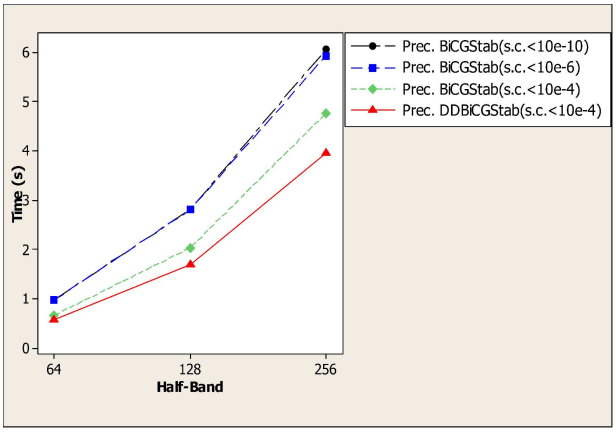


Figure 4.5: *Performance on 8 processors for different stopping criteria on ASIC_680k*

The overall behavior of the different algorithms is shown in Fig. 4.6 for banded systems extracted from the symmetric matrix G3_circuit, see [6]. Here, we show the speed improvement, or deterioration, compared to ScaLapack as the number of processors increase from 2 to 32. We note that: (i) ScaLapack outperforms LAPACK when the number of processors is 4 or higher, (ii) depending on the effectiveness of the preconditioner, preconditioned CG and BiCGstab can outperform ScaLapack, and perform similarly to our un-preconditioned hybrid solver, and (iii) our preconditioned hybrid solver outperforms all the rest as well as obtain a much smaller final residual norm. These remarks are also valid for

our experiments with the non-symmetric matrices extracted from ASIC_680k matrix, see [6].

Finally, scalability of the algorithms for the symmetric and non-symmetric systems is shown in Fig. 4.7 & 4.8, respectively.

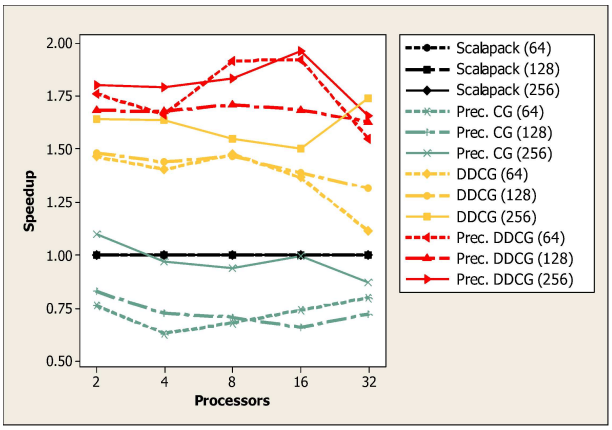


Figure 4.6: *Performance of CG (stopping criteria<10E-10), DDCG and ScaLapack on G3_circuit*

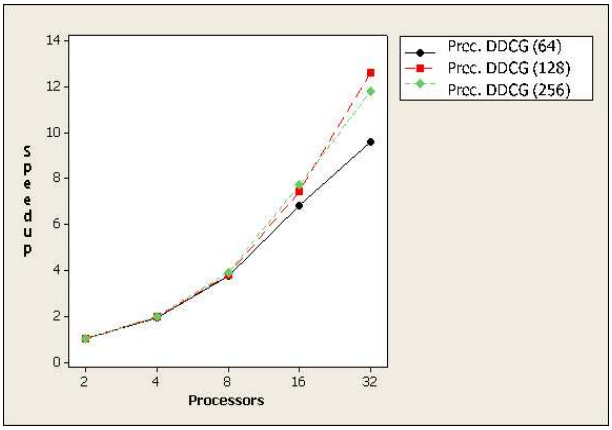


Figure 4.7: *Scalability of DDCG on G3_circuit for 32 processors.*

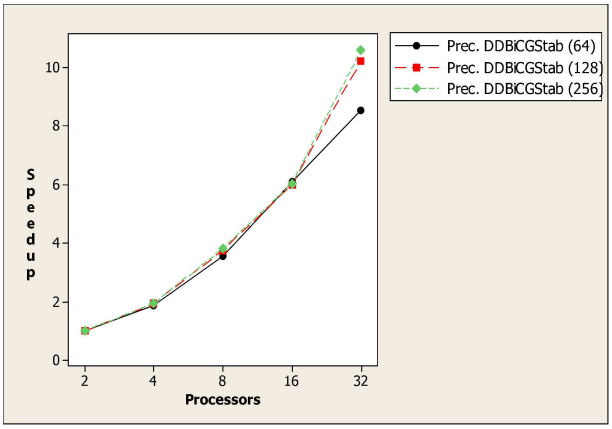


Figure 4.8: *Scalability of DDBiCGStab on ASIC_680k for 32 processors.*

4.7 Pseudocode

1. Partition the linear system according to (4.3) and (4.22) on p processors.
2. Compute $\mathbf{r}_0 = \mathbf{g}$ using (4.40) by solving p linear systems in the first term of (4.38).

3a. Start Modified Krylov method (initial guess $\mathbf{y}_0 = \mathbf{0}$ and resid. $\mathbf{r}_0$ have been computed)

Modified Prec. CG (see p. 15 in [5])

for i=1,2,...
  solve $\widetilde{M}\mathbf{z}_{i-1} = \mathbf{r}_{i-1}$
  $\rho_{i-1} = \mathbf{r}_{i-1}^T \mathbf{z}_{i-1}$
  if $i = 1$ then
    $\mathbf{p}_1 = \mathbf{z}_0$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
  end
  $\mathbf{q}_i = \text{modified\_multiply}(M, \mathbf{p}_i)$
  $\alpha_i = \rho_{i-1}/\mathbf{p}_i^T \mathbf{q}_i$
  $\mathbf{y}_i = \mathbf{y}_{i-1} + \alpha_i \mathbf{p}_i$
  $\mathbf{r}_i = \mathbf{r}_{i-1} - \alpha_i \mathbf{q}_i$
  check convergence
end

Modified Prec. BiCGStab (see p. 27 in [5])

for i=1,2,...
  $\rho_{i-1} = \tilde{\mathbf{r}}^T \mathbf{r}_{i-1}$ (if $\rho_{i-1} = 0$ failed)
  if i=1 then
    $\mathbf{p}_i = \mathbf{r}_{i-1}$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $\mathbf{p}_i = \mathbf{r}_{i-1} + \beta_{i-1}(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$
  end
  solve $\widetilde{M}\hat{\mathbf{p}} = \mathbf{p}_i$
  $\mathbf{v}_i = \text{modified\_multiply}(M, \hat{\mathbf{p}})$
  $\alpha_i = \rho_{i-1}/\tilde{\mathbf{r}}^T \mathbf{v}_i$, $\mathbf{s} = \mathbf{r}_{i-1} - \alpha_i \mathbf{v}_i$
  if $(||\mathbf{s}||_2 < \epsilon)$ then $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i \hat{\mathbf{p}}$ stop
  solve $\widetilde{M}\hat{\mathbf{s}} = \mathbf{s}$
  $\mathbf{t} = \text{modified\_multiply}(M, \hat{\mathbf{s}})$
  $\omega_i = \mathbf{t}^T \mathbf{s}/\mathbf{t}^T \mathbf{t}$, $\mathbf{r}_i = \mathbf{s} - \omega_i \mathbf{t}$
  $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i \hat{\mathbf{p}}_i + \omega_i \hat{\mathbf{s}}$
  check convergence (if $\omega_i \neq 0$ proceed)
end

3b. Notice that since $\mathbf{r}_i$, $\mathbf{y}_i$ are composed of $p-1$ blocked components, the same applies to $\mathbf{p}_i = [\mathbf{p}_1^{(i)^T}, \ldots, \mathbf{p}_{p-1}^{(i)^T}]^T$. Suppose that each of these block components is located on a separate processor (one processor remains idle) then modified_multiply steps are: exchange (processors $k = 2, \ldots, p-1$ send $\mathbf{p}_k^{(i)}$ to $k-1$ and $k = 1, \ldots, p-2$ send $\mathbf{p}_k^{(i)}$ to $k+1$) to construct right-hand-side in the second term of (4.38); solution of the system in the second term of (4.38); computation of the result using (4.41), preceded by communication where processors $k = 2, \ldots, p-1$ send $\bar{\mathbf{y}}_1^{(k)}$ to $k-1$.

4. Solve (4.1) by solving p independent linear systems (4.4), where $\mathbf{y}$ is known.

Chapter 5

Scalability of Parallel Programs

5.1 Introduction

Faced with the challenge of reducing the power consumption of conventional microprocessors while delivering increased performance, microprocessor vendors have leveraged increasing transistor counts to deliver multicore processors. Packaging multiple, possibly simpler cores, operating at lower voltage results in lower power consumption, a primary motivation for multicore processors. A byproduct of this is that parallelism becomes the primary driver for performance. Indeed, multicore designs with up to 16 cores are currently available, and some with up to 80 cores have been prototyped.

Multicore processors bring tremendous raw performance to the desktop. Harnessing this raw compute power in real programs, ranging from desktop applications to scientific codes is critical to their commercial and technological viability. Beyond the desktop, the packaging and power characteristics of multicore processors make it possible to integrate a large number of such processors into a parallel ensemble. This has the potential to realize the long-held vision of truly scalable parallel platforms. Indeed, systems with close to hundred thousand cores are likely to become available in the foreseeable future. The peak performance of such systems is anticipated to be several petaFLOPS.

The ability to build scalable parallel systems with very large number of processing cores puts forth a number of technical challenges. With the large component counts, the reliability of such systems is a major issue. System software support must provide a reliable logical machine view to the programmer, while supporting efficient program execution. Programming models and languages must allow a programmer to express concurrency and interactions in a manner that maximizes concurrency, minimizes overheads, while at the same time providing high-level abstractions of the underlying hardware. Runtime systems must continually monitor faults and application performance, while

providing diagnostic, prognostic, and remedial services. While all of these are important issues, in our belief, there is an even more fundamental question – *are there algorithms that can efficiently scale to such configurations, given hardware parameters, limitations, and overheads?* The answer to this question, and the methodology by which answers this question are critical.

Estimating the performance of a program on a larger hardware configuration from known performance on smaller configurations and smaller problem instances is generally referred to as scalability analysis. Indeed, this is precisely the problem we are faced with, when we try to assess applications and algorithms capable of efficiently utilizing emerging large-scale parallel platforms. Scalability analysis has been well-studied in the literature – with automated, analytical, and experimental methods proposed and investigated. Each class of methods has its limitations and advantages. Automated techniques [16, 40] work on realizations of algorithms (programs), as opposed to algorithms themselves. From this point of view, their role in guiding algorithm development is limited – since they are implementation based. Furthermore, automated techniques generally rely on parameterizing quanta of communication and computation, and using or simulations (Monte Carlo, discrete event simulation). For this reason, these methods have limited accuracy, or may themselves be computationally expensive.

Experimental approaches [3, 13, 27, 28, 79] to scalability analysis typically perform detailed quantification of the program instance and underlying platform, in order to generate a prediction model. These models can be used to predict performance beyond the observed envelope. As is the case with automated techniques, since these methods use realizations of algorithms, their ability to guide the algorithm development process is limited. Furthermore, because of the sensitive interaction between programs and underlying systems, the prediction envelope of such methods is often limited.

Analytical methods for scalability analysis rely on asymptotic estimates of basic metrics, such as parallel time and efficiency. Where available, such analytical estimates hold the potential for accurate scalability predictions, while providing guidance for algorithm and architecture development. For example, scalability analysis predicts the necessary network bandwidth/compute power in a system to efficiently scale Fast Fourier Transforms (FFTs) to large numbers of processors. The major drawback of these methods, of course, is that analytical quantification of parallel performance is not always easy. Even for simple algorithms, the presence of system (as opposed to programmer) controlled performance optimizations (caches and associated replacement policies are the simple examples) make accurate analytical quantification difficult.

In this chapter, we focus on analytical approaches to scalability analysis. Our choice is motivated by the fact that in relatively early stages of hardware development, such studies can effectively guide hardware design. Furthermore, we believe that for many applications, parallel algorithms scaling to very large configurations may not be the same as those that yield high efficiencies on smaller configurations. For these desired predictive properties, analytical modeling is critical.

There are a number of scaling scenarios, which may be suited to various application scenarios. In a large class of applications, the available memory limits the size of problem instance being solved. The question here is, given constraints on memory size, how does an algorithm perform? In other applications, for example, weather forecasting, there are definite deadlines on task completion. The question here becomes one of, what is the largest problem one can solve, given constraints on execution time, on a specific machine configuration. In yet other applications, the emphasis may be on efficiency. Here, one asks the question, what is the smallest problem instance I must solve on a machine so as to achieve desired execution efficiency. Each of these scenarios is quantified by a different scalability metric. We describe these metrics and their application to specific algorithms.

The rest of this chapter is organized as follows: in Section 5.2, we describe basic metrics of parallel performance, in Section 5.3, we describe various scalability metrics and how they can be applied to emerging parallel platforms. In Section 5.4, we discuss alternate parallel paradigms of task mapping for composite algorithms and how our scalability analysis applies to these scenarios.

5.2 Metrics of Parallel Performance

Throughout this chapter, we refer to a single processing core as a processor. Where we talk of a chip with multiple cores, we refer to it as a chip multiprocessor, or a multicore processor. We assume that a parallel processor (or ensemble) consists of p identical processing cores. The cores are connected through an interconnect. Exchanging a message of size m between cores incurs a time of $t_s + t_w m$. Here, t_s corresponds to the network latency and t_w the per-word transfer time determined by the network bandwidth. The communication model used here is a simplification, since it does not account for network congestion, communication patterns, overlapped access, and interconnect topology, among others. We define a parallel system as a combination of a parallel program and a parallel machine. We assume that the serial time of execution of an algorithm T_s is composed of W basic operations.

Perhaps the simplest and most intuitive metric of parallel performance is the parallel runtime, T_p. This is the time elapsed between initiation of the parallel program at the first processor to the completion of the program at the last processor. Indeed, an analytical expression for parallel time captures all the performance parameters of a parallel algorithm. The problem with parallel runtime is that it does not account for the resources used to achieve the execution time. Specifically, if one were to indicate that the parallel runtime of a program, which took 10s on a serial processor, is 2 seconds, we would have no way of knowing whether the parallel program (and associated algorithm) performs well or not. The obvious question w.r.t. parallel time is, how much lower it is compared to its serial counterpart. This speedup, S is defined as the ratio of the serial time T_s and the parallel time T_p. Mathematically, $S = T_s/T_p$. For the example above, the speedup is $10/2 = 5$.

While we have a sense of how much faster our program is compared to its serial counterpart, we still cannot estimate its performance, since we do not know how many resources were consumed to achieve this speedup. Specifically, if one were told that a parallel program achieved a speedup of 5, could we say anything about the performance of the program? For this reason, the speedup can be normalized with respect to the number of processors p to compute efficiency. Formally, efficiency $E = S/p$. In the example above, if the speedup of 5 is achieved using 8 processors, the efficiency is 5/8 or 0.625 (or 62.5%). If the same speedup is achieved using 16 processors, the corresponding efficiency is 0.312 (or 31.2%). Clearly, the algorithm can be said to perform better in the former case.

Efficiency, by itself, is not a complete indicator of parallel performance either. A given problem can be solved using several possible serial algorithms, which may be more or less amenable to parallel execution. For example, given a sparse linear system, we can solve it using simple Jacobi iterations. These are easily parallelizable and mostly involve near-neighbor communication (other than global dot products). On the other hand, the same system can be solved using a more powerful solver such as the Generalized Minimum Residual method, GMRES, see Yousef Saad, Iterative Methods for Sparse Linear Systems, second edition, SIAM, 2003, with an approximate inverse preconditioner. This method solves the problem much faster in the serial context, but is less amenable to parallelism, compared to the first algorithm. In such cases, one must be cautious relying simply on efficiency as a metric for performance.

To account for these issues, vendors and users sometimes rely on total aggregate FLOPS as a metric. Indeed, when we refer to a platform as being capable of ten petaFLOPS, it is in the context of some program and input, or the absolute peak performance, which may, or may not be achievable by any program. The most popular choice of a benchmark in this context is the Linpack benchmark. The Linpack benchmark has favorable data reuse and computation/communication characteristics, and therefore yields parallel performance closer to peak for typical parallel platforms. However, this begs the obvious question of what, if any, this implies for other applications that may not have the same data reuse characteristics. Indeed, Linpack numbers, referring to dense matrix operations are rarely indicative of machine performance on typical applications that have sparse kernels (PDE solvers), molecular dynamics type sparse interaction potentials, access workloads of large-scale data analysis kernels, or commercial server workloads. It is important to note that the oft-used sparse kernels often yield as low as 5% to 10% of peak performance on conventional processors because of limited data reuse.

In addition to the shortcomings mentioned above, basic metrics do not explicitly target scaling. Specifically, if the parallel time of program (algorithm) A for solving a problem is lower than that of algorithm B for solving the same problem, what does it say if the problem size is changed, the number of processors is increased, or if any of the computation/ communication parameters are varied. It turns out that a single sample point in the multidimensional performance space of parallel programs says nothing about the performance at

other points. This provides strong motivation for the development of scalability metrics.

5.3 Metrics for Scalability Analysis

We motivate scalability analysis through two examples, at two diverse ends of the spectrum. The first example relates to emerging large-scale platforms, and the second, to emerging scalable multicore desktop processors.

An important challenge to the parallel computing community currently is to develop a core set of algorithms and applications that can utilize emerging petascale computers. The number of cores in these platforms will be in the range of 100,000, in an energy efficient configuration. A number of important questions are posed in this context – (i) which, if any, of the current applications and algorithms are likely to be able to scale to such configurations, (ii) what fundamentally new algorithms will be required to scale to very large machine configurations, (iii) what are realizable architectural features that will enhance scaling characteristics of wide classes of algorithms, (iv) what are the memory and I/O requirements of such platforms, and (v) how can time-critical applications, such as weather forecasting, effectively utilize these platforms. Scalability analysis can answer many of these questions, while guiding algorithm and architecture design.

At the other end of the spectrum, desktop software vendors are grappling with questions of how to develop efficient software with meaningful development and deployment cycles. Specifically, will software, ranging from operating systems (Windows Vista, Linux, etc.) to desktop applications (word processing, media applications) be able to use 64 cores or beyond, likely to become available in high-end desktops and engineering workstations in a 5-year timeframe. The difficulty associated with this is that such platforms do not exist currently, therefore, one must design algorithms and software that can scale to future platforms. Again, scalability analysis can guide algorithm and software development in fundamental ways.

The general theme of these two motivating examples is that often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain larger number of processing elements. Whereas code development is simplified by using scaled-down versions of the machine and the problem, their performance and correctness is much more difficult to establish based on scaled-down systems.

Why is performance extrapolation so difficult? Consider three parallel algorithms for computing an n-point Fast Fourier Transform (FFT) on 64 processing cores. Figure 5.1 illustrates speedup as the value of n is increased to 18K. Keeping the number of processing cores constant, at smaller values of n, one would infer from observed speedups that binary exchange and 3-D transpose algorithms are the best. However, as the problem is scaled up to 18 K points and

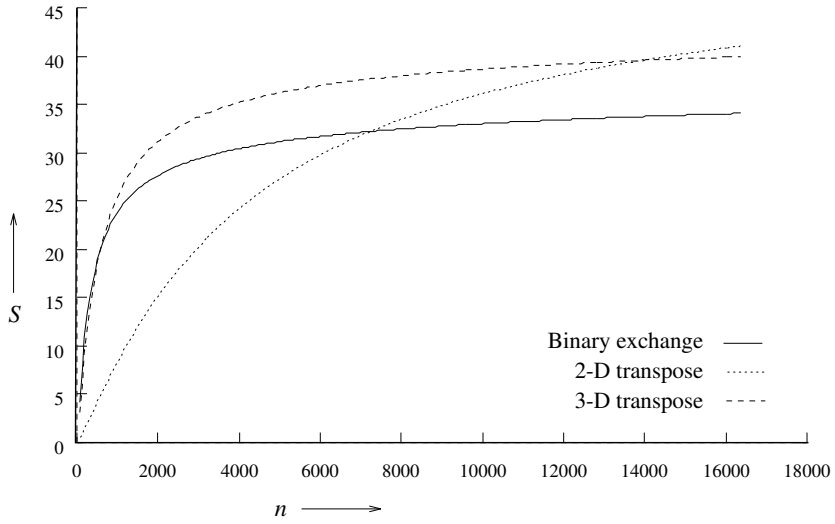


Figure 5.1: *A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 cores with* $t_c = 2ns$, $t_w = 4ns$, $t_s = 25ns$, *and* $t_h = 2ns$.

beyond, it is evident from Figure 5.1 that the 2-D transpose algorithm yields best speedup [39].

Similar results can be shown relating to the variation in number of processing cores as the problem size is held constant. Unfortunately, such parallel performance traces are the norm as opposed to the exception, making performance prediction based on limited observed data very difficult.

5.3.1 Scaling characteristics of parallel programs

The parallel runtime of a program, summed across all processing cores, (pT_p), consists of essential computation which would be performed by its serial counterpart (T_s), and overhead incurred in parallelization (T_o). Note that we use T_o to denote the total (or parallel) overhead summed across all processors. Formally, we write this as:

$$pT_p = T_s + T_o \tag{5.1}$$

We also know that the efficiency of a parallel program can be written as:

$$E = \frac{S}{p} = \frac{T_s}{pT_p}$$

Using the expression for parallel overhead (Equation 5.1), we can rewrite this expression as

$$E = \frac{1}{1 + \frac{T_o}{T_s}}. \tag{5.2}$$

The total overhead function T_o is an increasing function of p [31, 32]. This is because every program must contain some serial component. If this serial component of the program takes time t_{serial}, then during this time all the other processing elements must be idle. This corresponds to a total overhead function of $(p-1) \times t_{serial}$. Therefore, the total overhead function T_o grows at least linearly with p. Furthermore, due to communication, idling, and excess computation, this function may grow super-linearly in the number of processing cores. Equation 5.2 gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e., the value of T_s remains constant), as we increase the number of processing elements, T_o increases. In this scenario, it is clear from Equation 5.2 that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processing cores for a given problem size is common to all parallel programs. Often, in our quest for ever more powerful machines, this fundamental insight is lost – namely that the same problem instances that we solve on todays computers are unlikely to yield meaningful performance on emerging highly-parallel platforms.

Let us investigate the effect of increasing the problem size keeping the number of processing cores constant. We know that the total overhead function T_o is a function of both problem size T_s and the number of processing elements p. In many cases, T_o grows sub-linearly with respect to T_s. In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processing elements constant. Indeed, when demonstrating parallel performance on large machine configurations, many researchers assume that the problem is scaled in proportion to the number of processors. This notion of experimentally scaling problem size to maintain *constant computation per processor* is referred to as *scaled speedup* [41, 42, 43], and is discussed in Section 5.3.7. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processing elements simultaneously. This ability to maintain efficiency constant by simultaneously increasing the number of processing cores and the size of the problem is essential to utilizing scalable parallel platforms. Indeed, a number of parallel systems exhibit such characteristics. We call such systems *scalable* parallel systems. The *scalability* of a parallel system is a measure of its capacity to increase performance (speedup) in proportion to the number of processing cores. It reflects a parallel system's ability to utilize increasing processing resources effectively.

5.3.2 The isoefficiency metric of scalability

We summarize our discussion in the section above in the following two observations:

1. For a given problem size, as we increase the number of processing cores, the overall efficiency of the parallel system goes down. This phenomenon is common to all parallel systems.

2. In many cases, the efficiency of a parallel system increases if the problem

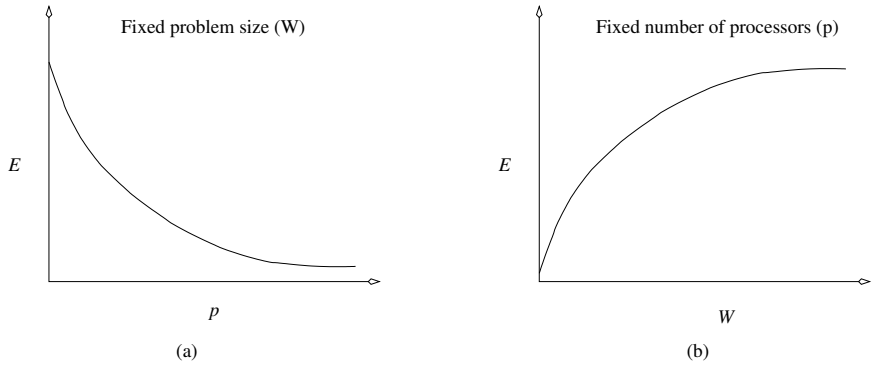


Figure 5.2: *Variation of efficiency: (a) as the number of processing elements is in creased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.*

size is increased while keeping the number of processing cores constant.

These two phenomena are illustrated in Figure 5.2(a) and (b), respectively. Following from these two observations, we define a scalable parallel system as one in which the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is also increased.

An important question that follows naturally is, what is the rate at which the problem size must be increased in order to keep efficiency constant. This rate is critical for a number of reasons – primarily, this rate determines the rate at which the total memory in the system must be increased. If problem size is linear in memory size, as is the case for a number of algorithms (applications), and the rate of increase in problem size is super-linear, this implies that the total memory in the parallel machine must increase super-linearly. This is a critical observation. Conversely, if the increase in system memory is sublinear with respect to the number of cores, one can trivially show that the parallel efficiency will decline. The relation between problem size, memory size, and efficiency, is a critical determinant of the scaling characteristics of parallel systems. A second implication of increasing problem size is that a super-linear increase results in an increasing (parallel) time to solution. For applications with constraints on time to solution (a classic example is in short-term weather forecasting), this may eventually limit growth in problem size. We discuss these three issues, namely, increase in problem size to maintain efficiency, impact of memory constraints, and impact of parallel solution time constraints separately.

For different parallel systems, the problem size must increase at different rates in order to maintain a fixed efficiency as the number of processing elements is increased. This rate is determined by the overheads incurred in parallelization. A lower rate is more desirable than a higher growth rate in problem size. This is because it allows us to extract good performance on smaller problems, and

consequently, improved performance on even larger problem instances. We now investigate metrics for quantitatively determining the degree of scalability of a parallel system. We start by formally defining the Problem Size.

Problem Size When analyzing parallel systems, we frequently encounter the notion of the size of the problem being solved. Thus far, we have used the term *problem size* informally, without giving a precise definition. A naive way to express problem size is as a parameter of the input size; for instance, n in case of a matrix operation involving $n \times n$ matrices. A drawback of this definition is that the interpretation of problem size changes from one problem to another. For example, doubling the input size results in an eight-fold increase in the execution time for matrix multiplication and a four-fold increase for matrix addition (assuming that the conventional $\Theta(n^3)$ algorithm is the best matrix multiplication algorithm, and disregarding more complicated algorithms with better asymptotic complexities). Using memory size as a measure of problem size has a similar drawback. In this case, the memory size associated with the dense matrices is $\Theta(n^2)$, which is also the time for, say, matrix addition and matrix-vector multiplication, but not for matrix-matrix multiplication.

A consistent definition of the size or the magnitude of the problem should be such that, regardless of the problem, doubling the problem size always means performing twice the amount of computation. Therefore, we choose to express problem size in terms of the total number of basic operations required to solve the problem. By this definition, the problem size is $\Theta(n^3)$ for $n \times n$ matrix multiplication (assuming the conventional algorithm) and $\Theta(n^2)$ for $n \times n$ matrix addition. In order to keep it unique for a given problem, we define *problem size* as the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing core. Since it is defined in terms of sequential time complexity, problem size W is a function of the size of the input.

Without loss of generality, we often assume that it takes unit time to perform one basic computation step of an algorithm. This assumption does not impact the analysis of any parallel system because the other hardware-related constants, such as message startup time, per-word transfer time, and per-hop time, can be normalized with respect to the time taken by a basic computation step. With this assumption, the problem size W is equal to the serial runtime T_s of the fastest known algorithm to solve the problem on a sequential computer.

The isoefficiency function

Parallel execution time can be expressed as a function of problem size, overhead function, and the number of processing elements. We can write parallel runtime as:

$$T_p = \frac{W + T_o(W, p)}{p} \tag{5.3}$$

The resulting expression for speedup is

$$\begin{aligned} S &= \frac{W}{T_p} \\ &= \frac{Wp}{W + T_o(W,p)}. \end{aligned} \tag{5.4}$$

Finally, we write the expression for efficiency as

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W,p)} \\ &= \frac{1}{1 + T_o(W,p)/W}. \end{aligned} \tag{5.5}$$

In Equation 5.5, if the problem size is kept constant and p is increased, the efficiency decreases because the total overhead T_o increases with p. If W is increased keeping the number of processing elements fixed, then for scalable parallel systems, the efficiency increases. This is because T_o grows slower than $\Theta(W)$ for a fixed p. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.

For different parallel systems, W must be increased at different rates with respect to p in order to maintain a fixed efficiency. For instance, in some cases, W might need to grow as an exponential function of p to keep the efficiency from dropping as p increases. Such parallel systems are poorly scalable. The reason is that on these parallel systems it is difficult to obtain good speedups for a large number of processing elements unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly scalable. That is because it can easily deliver speedups proportional to the number of processing elements for reasonable problem sizes.

For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio T_o/W in Equation 5.5 is maintained at a constant value. For a desired value E of efficiency,

$$\begin{aligned} E &= \frac{1}{1 + T_o(W,p)/W}, \\ \frac{T_o(W,p)}{W} &= \frac{1-E}{E}, \\ W &= \frac{E}{1-E} T_o(W,p). \end{aligned} \tag{5.6}$$

Let $K = E/(1-E)$ be a constant depending on the efficiency to be maintained. Since T_o is a function of W and p, Equation 5.6 can be rewritten as

$$W = KT_o(W,p). \tag{5.7}$$

From Equation 5.7, the problem size W can usually be obtained as a function of p by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as p increases. We call this function the *isoefficiency function* of the parallel system. The isoefficiency function determines the ease with which a parallel system can maintain constant efficiency, and hence achieve speedups increasing in proportion to the number of processing elements. A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processing elements, indicating that the parallel system is highly scalable. However, a large isoefficiency function indicates a poorly scalable parallel system. The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased.

Isoefficiency function of computing an n-point FFT. One parallel formulation of FFT, called the Binary Exchange Algorithm, computes an n-point FFT on a p processor machine (with $O(p)$, or full bisection bandwidth) in time:

$$T_p = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p$$

Recall here that t_c refers to unit computation, t_s to the message startup time, and t_w to the per-word message transfer time. Recall also that the problem size, W, for an n-point FFT is given by:

$$W = n \log n$$

The efficiency, E, of this computation can be written as:

$$E = \frac{T_s}{pT_p} = \frac{t_c n \log n}{t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p}$$

Through simple algebraic manipulations, we can rewrite the above expression as:

$$\frac{t_s p \log p}{t_c n \log n} + \frac{t_w log p}{t_c \log n} = \frac{1 - E}{E}$$

If efficiency E is held constant, the right hand side is constant. The equality can be forced in the asymptotic sense by forcing each of the terms in the left hand side to be constant, individually. For the first term, we have,

$$\frac{t_s p \log p}{t_c n \log n} = \frac{1}{K},$$

for constant K. This implies that the corresponding isoefficiency term is given by $W \approx n \log n \approx p \log p$. For the second term, we have,

$$\frac{t_w \log p}{t_c \log n} = \frac{1}{K},$$

$$\log n = K\frac{t_w}{t_c}\log p,$$

$$n = p^{Kt_w/t_c},$$

$$W = n\log n = K\frac{t_w}{t_c}p^{Kt_w/t_c}.$$

In addition to these two isoefficiency terms resulting from communication overhead, since the Binary Exchange algorithm can only use $p = n$ processors, there is an isoefficiency term resulting from concurrency. This term is given by $W = n\log n = p\log p$.

The combined isoefficiency from the three terms is the dominant of the three. If $Kt_w < t_c$ (which happens when we have very high bandwidth across the processing cores), the concurrency and startup terms dominate, and we have an isoefficiency of $p\log p$. Otherwise, the isoefficiency is exponential, and given by $W \approx p^{Kt_w/t_c}$. This observation demonstrates the power of isoefficiency analysis in quantifying both scalability of the algorithm, as well as desirable features of the underlying platform. Specifically, if we would like to use the Binary Exchange algorithm for FFT on large platforms, the balance of the machine should be such that $Kt_w < t_c$ (i.e., it must have sufficient per-processor bisection bandwidth).

The isoefficiency function [38] captures, in a single expression, the characteristics of a parallel algorithm as well as the parallel platform on which it is implemented. Isoefficiency analysis enables us to predict performance on larger number of processing cores from observed performance on a smaller number of cores. Detailed isoefficiency analysis can also be used to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processing cores and communication channels. In many cases, isoefficiency analysis can be used even for parallel algorithms for which we cannot derive a value of parallel runtime, by directly relating the total overhead to the problem size and number of processors [39].

5.3.3 Cost-optimality and the isoefficiency function

A parallel system is cost-optimal if the product of the number of processing cores and the parallel execution time is proportional to the execution time of the fastest known sequential algorithm on a single processing element. In other words, a parallel system is cost-optimal if and only if

$$pT_p = \Theta(W). \tag{5.8}$$

Substituting the expression for T_p from the right-hand side of Equation 5.3, we get the following:

$$\begin{aligned} W + T_o(W,p) &= \Theta(W) \\ T_o(W,p) &= O(W) \qquad (5.9) \\ W &= \Omega(T_o(W,p)) \qquad (5.10) \end{aligned}$$

Equations 5.9 and 5.10 suggest that a parallel system is cost-optimal if and only if its overhead function does not asymptotically exceed the problem size. This is very similar to the condition specified by Equation 5.7 for maintaining a fixed efficiency while increasing the number of processing elements in a parallel system. If Equation 5.7 yields an isoefficiency function $f(p)$, then it follows from Equation 5.10 that the relation $W = \Omega(f(p))$ must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

5.3.4 A lower bound on the isoefficiency function

We discussed earlier that a smaller isoefficiency function indicates higher scalability. Accordingly, an ideally-scalable parallel system must have the lowest possible isoefficiency function. For a problem consisting of W units of work, no more than W processing elements can be used cost-optimally; additional processing cores will be idle. If the problem size grows at a rate slower than $\Theta(p)$ as the number of processing elements increases, the number of processing cores will eventually exceed W. Even for an ideal parallel system with no communication, or other overhead, the efficiency will drop because processing elements added beyond $p = W$ will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\Theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\Theta(p)$.

This trivial lower bound implies that the problem size must grow at-least linearly, or, consequently, the problem size per processor must be at-least constant. For problems where memory size and problem size, W are linearly related, the memory per processing core must increase linearly to maintain constant efficiency.

5.3.5 The degree of Concurrency and the Isoefficiency Function

A lower bound of $\Omega(p)$ is imposed on the isoefficiency function of a parallel system by the number of operations that can be performed concurrently. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*. The degree of concurrency is a measure of the number of operations that an algorithm can perform in parallel for a problem of size W; it is independent of the parallel architecture. If $C(W)$ is the degree of concurrency of a parallel algorithm, then for a problem of size W, no more than $C(W)$ processing elements can be employed effectively.

Effect of concurrency on isoefficiency function. Consider solving a system of n equations in n variables by using Gaussian elimination. The total amount of computation is $\Theta(n^3)$. But the n variables must be eliminated one after the other (assuming pivoting), and eliminating each variable requires $\Theta(n^2)$ computations. Thus, at most $\Theta(n^2)$ processing elements can be kept busy at any time. Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$

is $\Theta(W^{2/3})$ and at most $\Theta(W^{2/3})$ processing elements can be used efficiently. On the other hand, given p processing cores, the problem size should be at least $\Omega(p^{3/2})$ to use them all. Thus, the isoefficiency function of this computation due to concurrency is $\Theta(p^{3/2})$.

The isoefficiency function due to concurrency is optimal (that is, $\Theta(p)$) only if the degree of concurrency of the parallel algorithm is $\Theta(W)$. If the degree of concurrency of an algorithm is less than $\Theta(W)$, then the isoefficiency function due to concurrency is worse (that is, greater) than $\Theta(p)$. In such cases, the overall isoefficiency function of a parallel system is given by the maximum of the isoefficiency functions due to concurrency, communication, and other overheads.

5.3.6 Scaling properties and parallel benchmarks

At this point, we can further investigate why dense linear algebra kernels are frequently used as benchmarks for parallel systems. Consider the example of matrix multiplication – the first thing to note is that it has significant data reuse, even in the serial sense ($\Theta(n^3)$ operations on $\Theta(n^2)$ data). The parallel runtime of a commonly used matrix multiplication algorithm (Cannon's algorithm) is given by:

$$T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}$$

The corresponding isoefficiency is given by $W = p^{1.5}$. The favorable memory to FLOPS ratio of matrix multiplication, combined with the relatively favorable isoefficiency function makes this an ideal parallel benchmark. Specifically, if $M \approx n^2$ represents the memory requirement, since $W \approx n^3$, we have $W \approx M^{1.5}$. Substituting in the expression for isoefficiency, we have, $W \approx M^{1.5} \approx p^{1.5}$. From this, we can see that we can keep efficiency constant with $M \approx p$. In other words, the increase in problem size to hold efficiency constant requires only a linear increase in total memory, or constant memory per processor.

The relatively benign memory requirement and their high processor utilization makes dense kernels, such as matrix-matrix multiplication and linear direct solvers ideal benchmarks for large-scale parallel platforms. It is important to recognize the favorable characteristics of these benchmarks, since a number of real applications, for example, sparse solvers and molecular dynamics methods, do not have significant data reuse and the underlying computation is linear in memory requirement.

5.3.7 Other scalability analysis metrics

We have alluded to constraints on runtime and memory as being important determinants of scalability. These constraints can be incorporated directly into scalability metrics. One such metric, called Scaled Speedup, increases the problem size linearly with the number of processing cores [41, 42, 43, 74, 80]. If the scaled-speedup curve is close to linear with respect to the number of processing cores, then the parallel system is considered scalable. This metric is related

to isoefficiency if the parallel algorithm under consideration has linear or near-linear isoefficiency function. In this case the scaled-speedup metric provides results very close to those of isoefficiency analysis, and the scaled-speedup is linear or near-linear with respect to the number of processing cores. For parallel systems with much worse isoefficiencies, the results provided by the two metrics may be quite different. In this case, the scaled-speedup versus number of processing cores curve is sublinear.

Two generalized notions of scaled speedup have been investigated. They differ in the methods by which the problem size is scaled up with the number of processing elements. In one method, the size of the problem is increased to fill the available memory on the parallel computer. The assumption here is that aggregate memory of the system increases with the number of processing cores. In the other method, the size of the problem grows with p subject to a bound on parallel execution time. We illustrate these using an example:

Memory and time-constrained scaled speedup for matrix-vector products. The serial runtime of multiplying a matrix of dimension $n \times n$ with a vector is n^2 (with normalized unit execution time). The corresponding parallel runtime using a simple parallel algorithm is given by:

$$T_p = \frac{n^2}{p} + \log p + n$$

and the speedup S is given by:

$$S = \frac{n^2}{\frac{n^2}{p} + \log p + n} \tag{5.11}$$

The total memory requirement of the algorithm is $\Theta(n^2)$. Let us consider the two cases of problem scaling. In the case of memory constrained scaling, we assume that the memory of the parallel system grows linearly with the number of processing cores, i.e., $m = \Theta(p)$. This is a reasonable assumption for most current parallel platforms. Since $m = \Theta(n^2)$, we have $n^2 = c \times p$, for some constant c. Therefore, the scaled speedup S' is given by:

$$S' = \frac{c \times p}{\frac{c \times p}{p} + \log p + \sqrt{c \times p}}$$

or

$$S' = \frac{c_1 p}{c_2 + c_3 \log p + c_4 \sqrt{p}}.$$

In the limiting case, $S' = O(\sqrt{p})$.

In the case of time constrained scaling, we have $T_p = O(n^2/p)$, and since this is constrained to be constant, we have $n^2 = O(p)$. We notice that this case is identical to the memory constrained case. This happened because the memory and runtime of the algorithm are asymptotically identical.

Memory and time-constrained scaled speedup for matrix-matrix products. The serial runtime of multiplying two matrices of dimension $n \times n$ is n^3. The corresponding parallel runtime using a simple parallel algorithm is given by:

$$T_p = \frac{n^3}{p} + \log p + \frac{n^2}{\sqrt{p}}$$

and the speedup S is given by:

$$S = \frac{n^3}{\frac{n^3}{p} + \log p + \frac{n^2}{\sqrt{p}}} \tag{5.12}$$

The total memory requirement of the algorithm is $\Theta(n^2)$. Let us consider the two cases of problem scaling. In the case of memory constrained scaling, as before, we assume that the memory of the parallel system grows linearly with the number of processing elements, i.e., $m = \Theta(p)$. Since $m = \Theta(n^2)$, we have $n^2 = c \times p$, for some constant c. Therefore, the scaled speedup S' is given by:

$$S' = \frac{(c \times p)^{1.5}}{\frac{(c \times p)^{1.5}}{p} + \log p + \frac{c \times p}{\sqrt{p}}} = O(p)$$

In the case of time constrained scaling, we have $T_p = O(n^3/p)$. Since this is constrained to be constant, $n^3 = O(p)$, or $n^3 = c \times p$ (for some constant c). Therefore, the time-constrained speedup S'' is given by:

$$S'' = \frac{c \times p}{\frac{c \times p}{p} + \log p + \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

This example illustrates that memory-constrained scaling yields linear speedup, whereas time-constrained speedup yields sublinear speedup in the case of matrix multiplication.

Serial Fraction f. The experimentally determined serial fraction f can be used to quantify the performance of a parallel system on a fixed-size problem. Consider a case when the serial runtime of a computation can be divided into a totally parallel and a totally serial component, i.e.,

$$W = T_{ser} + T_{par}.$$

Here, T_{ser} and T_{par} correspond to totally serial and totally parallel components. From this, we can write:

$$T_p = T_{ser} + \frac{T_{par}}{p}.$$

Here, we have assumed that all of the other parallel overheads such as excess computation and communication are captured in the serial component T_{ser}. From these equations, it follows that:

$$T_p = T_{ser} + \frac{W - T_{ser}}{p} \tag{5.13}$$

The serial fraction f of a parallel program is defined as:

$$f = \frac{T_{ser}}{W}.$$

Therefore, from Equation 5.13, we have:

$$\begin{aligned} T_p &= f \times W + \frac{W - f \times W}{p} \\ \frac{T_p}{W} &= f + \frac{1-f}{p} \end{aligned}$$

Since $S = W/T_p$, we have

$$\frac{1}{S} = f + \frac{1-f}{p}.$$

Solving for f, we get:

$$f = \frac{1/S - 1/p}{1 - 1/p}. \tag{5.14}$$

It is easy to see that smaller values of f are better since they result in higher efficiencies. If f increases with the number of processing elements, then it is an indicator of rising communication overhead, and thus an indicator of poor scalability.

Serial component of the matrix-vector product. From Equations 5.14 and 5.11, we have

$$f = \frac{\frac{\frac{n^2}{p} + \log p + n}{n^2}}{1 - 1/p} \tag{5.15}$$

Simplifying the above expression, we get

$$\begin{aligned} f &= \frac{p \log p + np}{n^2} \times \frac{1}{p-1} \\ f &\approx \frac{\log p + n}{n^2} \end{aligned}$$

It is useful to note that the denominator of this equation is the serial runtime of the algorithm and the numerator corresponds to the overhead in parallel execution.

A comprehensive discussion of various scalability and performance measures can be found in the survey by Kumar and Gupta [50]. While over ten years old, the results cited in this survey and their applications are particularly relevant now, as scalable parallel platforms are finally being realized.

5.4 Heterogeneous Composition of Applications

Many applications are composed of multiple steps, with differing scaling characteristics. For example, in commonly used molecular dynamics techniques, the

potentials on particles are evaluated as sums of near and far-field potentials. Near field potentials consider particles in a localized neighborhood, typically resulting in a linear algorithmic complexity. Far-field potentials, along with periodic boundary conditions are evaluated using Ewald summations, which involve computation of an FFT. Short-range potentials can be evaluated in a scalable manner on most platforms, since they have favorable communication to computation characteristics (communication is required only for particles close to the periphery of a processing core's sub-domain). FFTs, on the other hand have more exacting requirement on network bandwidth, as demonstrated earlier in this chapter.

Traditional parallel paradigms attempt to distribute computation associated with both phases across all processing cores. However, since FFTs generally scale worse than the short-range potentials, it is possible to use a heterogeneous partition of the platform – with one partition working on short-range potentials, and the other on computing the FFT. The results from the two are combined to determine total potential (and force) at each particle.

Scalability analysis is a critical tool in understanding and developing heterogeneous parallel formulations. The tradeoffs of sizes of heterogeneous partitions, algorithm scaling, and serial complexities are critical determinants of overall performance.

Chapter 6

Summary and Conclusion

We presented an effective parallel algorithm, SPIKE, for solving banded linear systems. Comparisons against the corresponding solvers in ScaLapack prove that our algorithm used as a direct solver is faster than ScaLapack, and if one is satisfied with reasonable relative residuals, SPIKE can be used as a very effective preconditioner of an iterative scheme such as BiCGstab. We demonstrated the effectiveness of SPIKE on varying numbers of processors with increasing bandwidth, as well as its favorable scalability on a large number of processors on the IBM-SP. Clearly, in deciding on the original partitioning of the system, a balance must be achieved between the computational cost of solving the subproblems and the communication overhead. For parallel architectures in which the communication costs are relatively high, the optimal implementation of the SPIKE algorithm will favor fewer partitions to reduce the communication overhead. On the other hand, on parallel architectures with relatively fast interconnects and in which each CPU has vector processing capabilities, the SPIKE algorithm can be further modified to capitalize on trade-offs between parallel and vector processing.

Three members of the SPIKE family of schemes have been described: (i) the Recursive SPIKE, (ii) the Truncated SPIKE and (iii) the SPIKE "on-the-fly". Each of these schemes uses a different solution strategy of the reduced system. The resulting reduced system can be solved either explicitly using a parallel recursive method, approximately, or implicitly using iterative methods. The Recursive and Truncated versions of the algorithm for handling both diagonally and non-diagonally dominant systems, have been implemented and compared with the corresponding solvers in ScaLapack for systems that are dense within the band. The speed improvements realized by SPIKE over ScaLapack are significant. The favorable scalability of SPIKE has also been demonstrated on the Intel-Xeon cluster. For banded systems that are sparse within the band, SPIKE "on-the-fly" with two-levels of parallelism exhibits good scalability.

We should note that the SPIKE environment is specifically well suited for recently introduced computational fluid dynamics techniques such as those in [75], [76], or [73]. In [75] the finite element grid is partitioned (statically or

dynamically) into "iterative" or "direct" groups, depending on which part of the grid poses a challenge to iterative schemes. Also, the two-level SPIKE scheme is ideally suited for the multi-scale solution technique presented in [76]. Finally, any member of the SPIKE family can be used for handling those linear systems that arise in the "solid-extension mesh moving technique", in [73], in regions of the grid with very small elements which are at risk of being "entangled" if classical iterative schemes (with "black-box" preconditioners) converge slowly.

We have also demonstrated that banded preconditioners can be attractive alternatives to ILU-type preconditioners, which might be desirable particularly for parallel solvers. Banded preconditioners offer high FLOP counts and concurrency in addition to desirable convergence properties for a range of problems. We identified the central role of matrix reordering in the performance of banded preconditioners. We proposed the use of weighted spectral ordering for this purpose. We demonstrated that, weighted bandwidth reduction based on spectral reordering, used in conjunction with efficient banded solvers is capable of significantly better performance than even optimized versions of ILU for a variety of problems, both in terms of convergence and time-to-solution.

We have also developed a parallel hybrid algorithm for the solution of banded linear systems that result from domain decomposition. The conditions that must be satisfied to guarantee that the balance system is symmetric positive definite or nonsingular are also stated. Our hybrid algorithm can be viewed as solving a large system of linear equations on one hand, and performing the LU-factorization of several smaller independent linear systems once, but applying the forward and backward sweeps several times, on the other. We describe how the banded system can be "torn" into overlapped smaller systems that can be solved independently under certain constraints, giving rise to a much smaller balance system that is not formed explicitly. Further, we showed how this system can be solved via a modified preconditioned Krylov subspace method with a preconditioner that takes advantage of the decay property of the inverse of banded systems. Finally, numerical experiments show that our proposed algorithm outperforms sequential LAPACK, parallel ScaLapack, preconditioned CG and BiCGStab for symmetric and non-symmetric banded systems, respectively.

Finally, we have demonstrated the power of scalability analysis, and more generally, analytical modeling. It is not always possible to do such modeling under tight bounds. Analytical modeling must account for finite resources and parameters such as line sizes, replacement policies, and other related intricacies. Clearly such analysis, if performed precisely, becomes specific to problem instance and platform, and does not generalize. While aforementioned limitations apply specifically to small-to-moderate sized systems, larger aggregates (terascale and beyond) are generally programmed through explicit message transfers. This is accomplished either through MPI-style messaging, or explicit (put-get) or implicit (partitioned global arrays) one-way primitives. In such cases, even with loose bounds on performance of individual multicore processors, we can establish tight bounds on scalability of the complete parallel system. Analytical modeling and scalability analysis can be reliably applied to such systems as well.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Apl. Math., Philadelphia, PA, third edition, (1999).

[2] P. Arbenz, A. Cleary, J. Dongarra and M. Hegland A Comparison of Parallel Solvers for Diagonally Dominant and General Narrow-Banded Linear Systems. *Parallel and Distributed Computing Practices* Vol. 2, pp. 385-400, (1999).

[3] Brian Armstrong, Hansang Bae, Rudolf Eigenmann, Faisal Saied, Mohamed Sayeed, and Yili Zheng. Hpc benchmarking and performance evaluation with realistic applications. In *Proceedings of Benchmarking Workshop*, 2006.

[4] S. T. Barnard, A. Pothen, and H. Simon, *A spectral algorithm for envelope reduction of sparse matrices*, Numerical Linear Algebra with Applications, 2 (1995), pp. 317–334.

[5] R. Barrett, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1994.

[6] M. Benzi, D. B. Szyld, and A. van Duin, *Orderings for incomplete factorization preconditioning of nonsymmetric problems*, SIAM Journal on Scientific Computing, 20 (1999), pp. 1652–1670.

[7] M. Benzi, J. C. Haws, and M. Tuma, *Preconditioning highly indefinite and nonsymmetric matrices*, SIAM J. Sci. Comput., 22 (2000), pp. 1333–1353.

[8] M. Benzi, G. Golub and J. Liensen, *Numerical Solution of Saddle Point Problems* Acta Numerica, (2005), pp. 1-137.

[9] M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Philippe and A. H. Sameh. *Parallel Algorithms on the CEDAR System* . In: *CONPAR 86, Lecture Notes in Computer Science* , W. Handler et. al., editors, Springer-Verlag, pp. 25-39, 1986.

[10] Michael Berry and Ahmed Sameh. *Multiprocessor Schemes for Solving Block Tridiagonal Linear Systems . International Journal of Supercomputer Applications* , Vol. 2, No. 3, pp. 37-57, 1988.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK: a linear algebra library for message-passing computers*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997), Philadelphia, PA, USA, 1997, Society for Industrial and Applied Mathematics, p. 15 (electronic).

[12] L.S Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Apl. Math., Philadelphia, PA, (1997).

[13] Shirley Browne, Jack Dongarra, and Kevin London. Review of performance analysis tools for MPI parallel programs. *NHSE Review*, 1998.

[14] P. K. Chan, M. D. F. Schlag, and J. Zien, *Spectral k-way ratio-cut partitioning and clustering*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13 (1994), pp. 1088–96.

[15] D. J. Kuck, S. C. Chen and A. H. Sameh, *Practical parallel band triangular system solvers*, ACM Transactions on Mathematical Software, 4 (1978), pp. 270–277.

[16] Nilesh Choudhury, Yogesh Mehta, Terry L. Wilmarth, Eric J. Bohm, and Laxmikant V. Kale. Scaling an optimistic parallel simulation of large-scale interconnection networks. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 591–600. Winter Simulation Conference, 2005.

[17] E. Chow and Y. Saad, *Experimental study of ILU preconditioners for indefinite matrices*, Journal of Computational and Applied Mathematics, 86 (1997), pp. 387–414.

[18] A. Cleary and J. Dongarra. Implemantation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems. *University of Tennessee Computer Science Technical Report*, UT-CS-97-358, (1997).

[19] J. M. Conroy, *Parallel Algorithms for the Solution of Narrow Banded Systems*, Applied Numerical Mathematics, 5 (1989), pp. 409-421.

[20] E. Cuthill and J. McKee, *Reducing the Bandwidth of Sparse Symmetric Matrices*, In Proceedings of the 1969 24th National Conf., (1969), pp. 157-172.

[21] T. Davis, *The University of Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/research/sparse/matrices/.

[22] S. Demko, W. F. Moss and P. W. Smith. Decay Rates for Inverses of Band Matrices. *Mathematics of Computation*, Vol. 43, No. 168, pp. 491-499 (1984).

[23] J. J. Dongarra and A. H. Sameh, *On Some Parallel Banded System Solvers*, Parallel Computing, 1 (1984), pp. 223-235.

[24] J. J. Dongarra and S. L. Johnson, *Solving Banded Systems on a Parallel Processor*, Parallel Computing, 5 (1988), pp. 219246.

[25] I. S. Duff, *On algorithms for obtaining a maximum transversal*, ACM Trans. Math. Softw., 7 (1981), pp. 315–330.

[26] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 20 (1999), pp. 889–901.

pp. 889–901.

[27] Rudolf Eigenmann (Editor). *Performance Evaluation and Benchmarking with Realistic Applications*. MIT Press, Cambrige, MA, 2001.

[28] T. Fahringer and S. Pllana. Performance prophet: A performance modeling and prediction tool for parallel and distributed programs. Technical Report 2005-08, AURORA Technical Report, 2005.

[29] M. Fiedler, *Algebraic Connectivity of Graphs*, Czechoslovak Mathematical Journal, 23 (1973), pp. 298-305.

[30] M. Fiedler, *A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Applications to Graph Theory*, Czechoslovak Mathematical Journal, 25 (1975), pp. 619-633.

[31] Horace P. Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1–20, 1989.

[32] Horace P. Flatt. Further applications of the overhead model for parallel systems. Technical Report G320-3540, IBM Corporation, Palo Alto Scientific Center, Palo Alto, CA, 1990.

[33] K. Gallivan, E. Gallopoulos and A. Sameh. *Cedar: An Experiment in Parallel Computing*, *Computer Mathematics and its Applications*, Vol. 1, No. 1, pp. 77-98, 1994.

[34] A. George, *Numerical Experiments Using Dissection Methods to solve n by n Grid Problems*, SIAM Journal on Numerical Analysis, 14 (1977), pp. 161-179.

[35] J. R. Gilbert and S. Toledo, *An assessment of incomplete-LU preconditioners for nonsymmetric linear systems*, Informatica (Slovenia), 24 (2000).

[36] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Third Edition, The John Hopkins University Press, 1996.

[37] G. Golub, A. Sameh and V. Sarin, *A Parallel Balance Scheme for Banded Linear Systems*, Numerical Linear Algebra with Appl., 8 (2001), pp. 297-316.

[38] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, 1 (3):12–21, 1993.

[39] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[40] D.A. Grove and P.D. Coddington. A performance modeling system for message-passing parallel programs. Technical Report DHPC-105, Department of Computer Science, Adelaide University, Adelaide, SA 5005, 2001.

[41] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[42] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[43] John L. Gustafson. The consequences of fixed time performance measurement. In *Proceedings of the 25th Hawaii International Conference on System Sciences: Volume III*, pages 113–124, 1992.

[44] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Comput., 16 (1995), pp. 452–469.

[45] HSL, *A collection of Fortran codes for large-scale scientific computation*, 2004. See http://www.cse.scitech.ac.uk/nag/hsl/.

[46] Y. Hu and J. Scott, *HSL_MC73: a fast multilevel Fiedler and profile reduction code*, Technical Report RAL-TR-2003-036, 2003.

[47] S. L. Johnson, *Solving Narrow Banded Systems on Ensemble Architectures*, ACM Transactions on Mathematical Software, 11 (1985), pp. 271-288.

[48] R. Kechroud, A. Soulaimani, and Y. Saad, *Preconditionning techniques for the solution of the helmholtz equation by the finite element method.*, in ICCSA (2), V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, eds., vol. 2668 of Lecture Notes in Computer Science, Springer, 2003, pp. 847–858.

[49] N. P. Kruyt, *A conjugate gradient method for the spectral partitioning of graphs*, Parallel Computing, 22 (1997), pp. 1493–1502.

[50] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994. Also available as Technical Report TR 91-18, Department of Computer Science Department, University of Minnesota, Minneapolis, MN.

[51] D. H. Lawrie and A. H. Sameh, *The computation and communication complexity of a parallel banded system solver*, ACM Trans. Math. Softw., 10 (1984), pp. 185–195.

[52] X. S. Li and J. W. Demmel, SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, Vol. 29, No. 2, pp. 110-140 (2003).

[53] W. Liu and A. H. Sherman, *Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices*, SIAM Journal on Numerical Analysis, 13 (1976), pp. 198-213.

[54] G. Lou, *Parallel Methods for Solving Linear Systems via Overlapping Decomposition* M.S. Thesis, University of Illinois - Urbana Champagne, 1989.

[55] U. Meier, *A Parallel Partition Method for Solving Banded Systems of Linear Equations*, Parallel Computing, 2 (1985), pp. 33-43.

[56] P.R. Amestoy, I. S. Duff, J.-Y L'Excellent and K. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling. *A SIAM Journal on Matrix Analysis and Applications*, Vol. 23, No. 21, pp. 15-49 (2001).

[57] M. Ng, *Fast iterative methods for symmetric sinc-Galerkin systems*, IMA J Numer Anal, 19 (1999), pp. 357–373.

[58] K. Gartner and O. Schenk, PARDISO: The current state and algorithmic goals for the future. *Proceedings of the Workshop on State-of-the-Art in Scientific Computing, PARA'04, (2004).*

[59] E. Polizzi and A. Sameh, *A parallel hybrid banded system solver: the SPIKE algorithm*, Parallel Computing, 32 (2006), pp. 177-194.

[60] E. Polizzi, A. Sameh, *SPIKE: A Parallel Environment for Solving Banded Linear Systems*, Computers and Fluids, 36 (2007), pp. 113-120.

[61] M. S. Reeves, D. C. Chatfield, and D. G. Truhlar, *Preconditioned complex generalized minimal residual algorithm for dense algebraic variational equations in quantum reactive scattering*, The Journal of Chemical Physics, 99 (1993), pp. 2739–2751.

[62] J. K. Reid and J. A. Scott, *Implementing hager's exchange methods for matrix profile reduction*, ACM Trans. Math. Softw., 28 (2002), pp. 377–391.

[63] Y. Saad, *SPARSKIT: A basic tool kit for sparse matrix computations*, Tech. Rep. 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.

[64] ——, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[65] A. H. Sameh. *Numerical Parallel Algorithms–A Survey* . In: *High Speed Computer and Algorithm Organization* , D. Kuck, D. Lawrie and A. Sameh, editors, Academic Press, pp. 207-228, 1977.

[66] A. H. Sameh and D. J. Kuck. *Parallel Direct Linear System Solvers–A Survey* . *IMACS (AICA)-GI-Symp. on Parallel Computers–Parallel Mathematics* , pp. 25-30, March, 1977.

[67] A. H. Sameh and D. J. Kuck. On Stable Parallel Linear System Solvers . *Journal of the ACM*, Vol. 25, pp. 81-91, 1978.

[68] A. H. Sameh. *On Two Numerical Algorithms for Multiprocessors. Proc. of NATO Adv. Res. Workshop on High-Speed Comp. (Series F: Computer and Systems Sciences, Vol. 7)* , Springer-Verlag, pp. 311-328, 1983.

[69] A. H. Sameh. *A Fast Poisson Solver on Multiprocessors* . In: *Elliptic Problem Solvers II* . Academic Press, pp. 175-186, 1984.

[70] A. Sameh. *Parallel Linear System Solvers . Proc. of the Conf. on Vector and Parallel Processors for Scientific Computation* , May 27-29, 1985.

[71] A. Sameh, and V. Sarin. *Hybrid Parallel Linear Solvers*, *International Journal of Computational Fluid Dynamics*, Vol 12, pp. 213-223, 1999.

[72] D. A. Spielman and S. Teng, *Spectral Partitioning Works: Planar Graphs and Finite Element Meshes*, Lin. Algebra and Appl., 421 (2007), pp. 284-305.

[73] K. Stein, T.E. Tezduyar and R. Benney, Automatic mesh update with the solid-extension mesh moving technique. *Computer Methods in Applied Mechanics and Engineering*, Vol. 193, pp. 2019-2032 (2004).

[74] X.-H. Sun and L. M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19:27–37, September 1993.

[75] T.E. Tezduyar and S. Sathe, Enhanced-approximation linear solution technique (EALST). *Computer Methods in Applied Mechanics and Engineering*, Vol. 193, pp. 2033-2049 (2004).

[76] T.E. Tezduyar and S. Sathe, Enhanced-discretization successive update method (EDSUM). *International Journal for Numerical Methods in Fluids*, Vol. 47, pp. 633-654 (2005).

[77] Z. Tong and A. Sameh, *On optimal banded preconditioners for the five-point laplacian*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 477–480.

[78] H. A. van der Vorst, *Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631–644.

[79] Vikram S. Adve, Rajive Bagrodia, James C. Browne, Ewa Deelman, Aditya Dubeb, Elias N. Houstis, John R. Rice, Rizos Sakellariou, David Sundaram-Stukel, Patricia T. Teller, and Mary K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *Software Engineering*, 26(11):1027–1048, 2000.

[80] Patrick H. Worley. Limits on parallelism in the numerical solution of linear PDEs. *SIAM Journal on Scientific and Statistical Computing*, 12:1–35, January 1991.

[81] S. J. Wright, *Parallel Algorithms for Banded Linear Systems*, SIAM Journal of Scientific and Statistical Computing, 12 (1991), pp. 824-842.

[82] J. Zhang, *On preconditioning Schur complement and Schur complement preconditioning*, Elect. Trans. Numer. Anal., 10 (2000), pp. 115–130.